PROCEEDINGS OF THE ELEVENTH ANNUAL NATIONAL CONFERENCE ON Ada TECHNOLOGY

MARCH 15-18, 1993

Sponsored By:
ANCOST, INC.

With Participation By:
United States Army
United States Navy
United States Air Force
United States Marine Corps
Ada Joint Program Office
Defense Information Systems Agency
Federal Aviation Administration
National Aeronautics & Space Administration

Academic Host:

PROCEEDINGS OF THE ELEVENTH ANNUAL NATIONAL CONFERENCE ON Ada TECHNOLOGY

Sponsored By:
ANCOST, INC.

With Participation By:
United States Army
United States Navy
United States Air Force
United States Marine Corps
Ada Joint Program Office
Defense Information Systems Agency
Federal Aviation Administration
National Aeronautics & Space Administration

Academic Host:
Virginia State University

Williamsburg Hilton--Williamsburg, VA

March 15-18, 1993

Approved for Public Release: Distribution Unlimited

ELEVENTH ANNUAL NATIONAL CONFERENCE ON Ada TECHNOLOGY CONFERENCE COMMITTEE 1992-1993

Executive Committee Chair:
MR. STEVE LAZEROWICH, Alsys

Treasurer:
MS. SUSAN MARKEL, TRW

Secretary:
MR. MURRAY KIRCH, Stockton State University

Immediate Past Chair:
MS. DEE GRAUMANN, GDE Systems, Inc.

Conference Chair:
MS. JUDITH M. GILES, Intermetrics, Inc.

Academic Outreach:
MR. JAMES WALKER, Network Solutions

Budget Committee Chair:
MR. MICHAEL SAPENTER, Telos Federal Systems

Policies, Procedures & By-Laws Chair:
DR. RICHARD KUNTZ, Monmouth College

Public Relations Chair:
MS. BEVERLY SAMPSON, DISA

Panels Chair:
MS. DEE GRAUMANN, GDE Systems, Inc.

Tutorial Chair:
MS. CHRISTINE BRAUN, GTE Federal Systems

Technical Program Chair:
MR. DANIEL E. HOCKING, Army Research Laboratory

Student Papers Chair:
DR. MARTIN BARRETT, Penn State University Harrisburg

MR. MIGUEL A. CARRIO, JR., MTM Engineering

MS. LUWANA S. CLEVER, Florida Institute of Technology

MR. NOEL GOYETTE, Computer Science Corporation

MS. JUDY GRIFFIN, Software Modernization Institute

DR. JAMES HOOPER, Marshall University

MR. GLEN HUGHES, Rational

DR. CHARLES LILLIE, SAIC

MR. EMMANUEL OMOJOKUN, Virginia State University

MS. LAURA VEITH, Rational

Conference Director:
MARJORIE Y. RISINGER, CMP, Rosenberg & Risinger

ADVISORY MEMBERS...

MR. CURRIE COLKET, US Navy

Maj. TOM CROAK, US Air Force

MR. JEFFREY HERMAN, US Army CECOM, SED

MR. DANIEL E. HOCKING, Army Research Laboratory

MR. HUET LANDRY, DISA Center for Standards

MR. E.V. (SKIP) SALTER, DISA

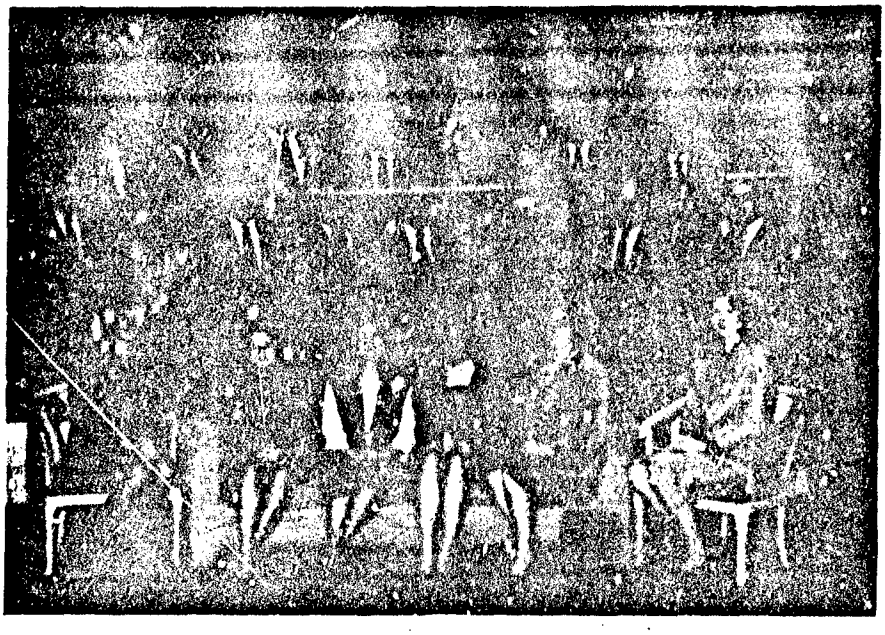
DR. JOHN SOLOMOND, AJPO

MR. CARRINGTON STEWART, NASA

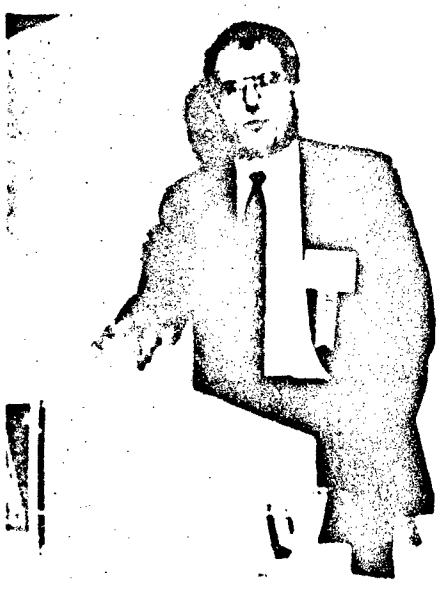
MS. CAROLYN STRANO, FAA

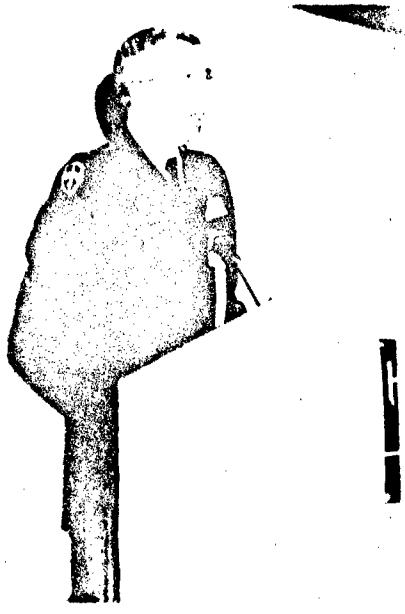
Maj. DAVID THOMPSON, USMC

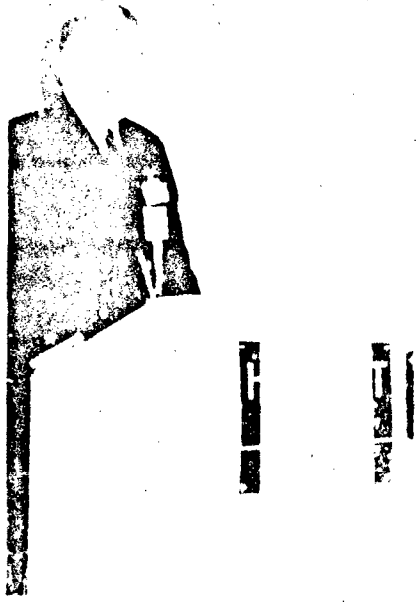
MS. KAY TREZZA, US Army CECOM, SED



Tenth Annual National Ada Conference Committee

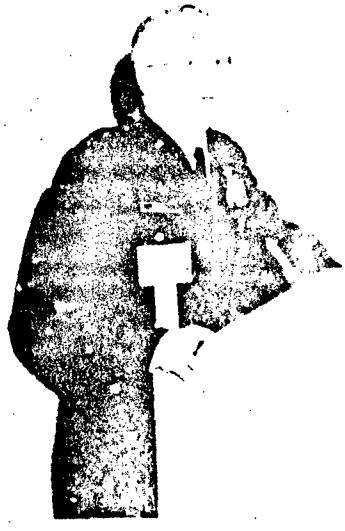




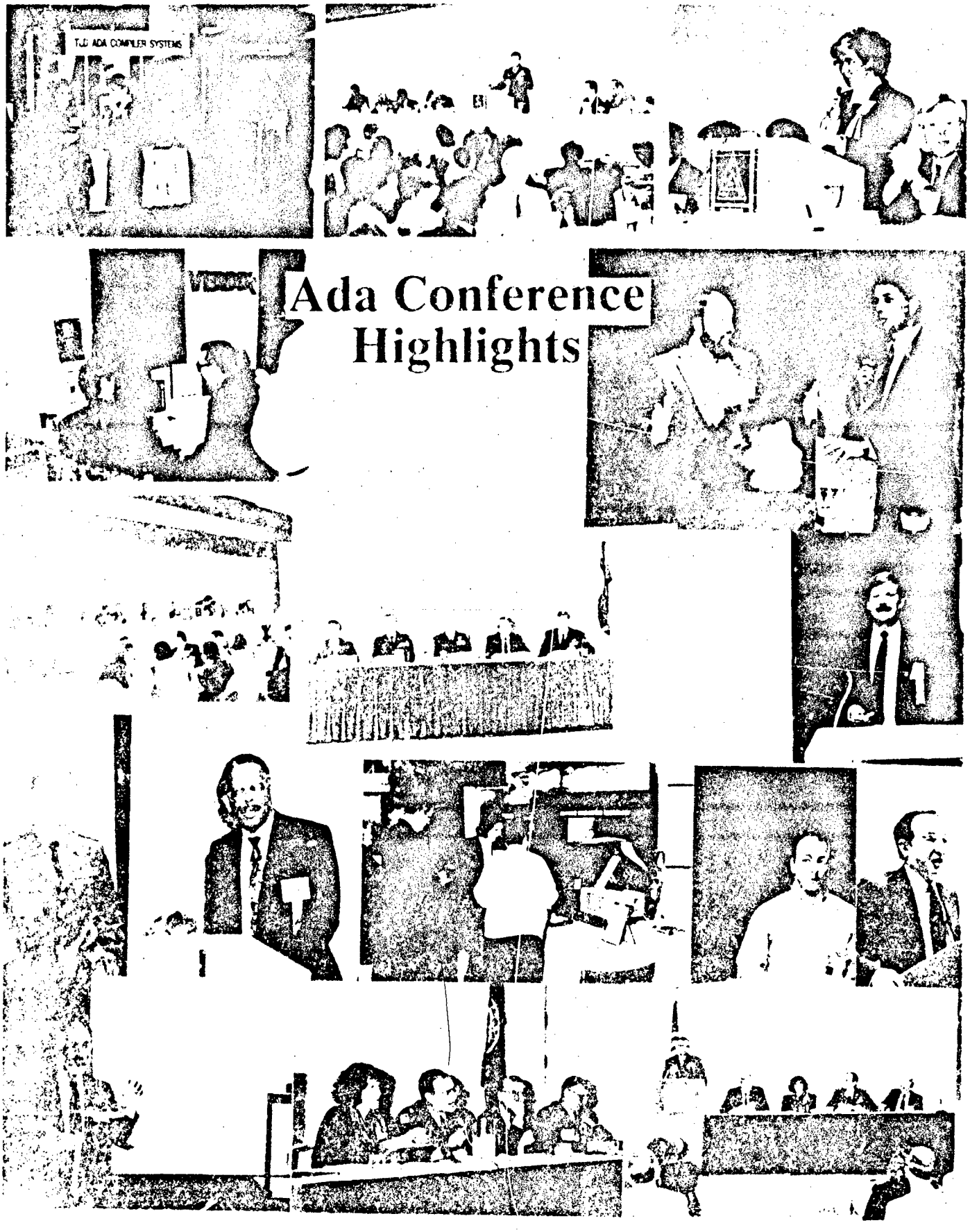


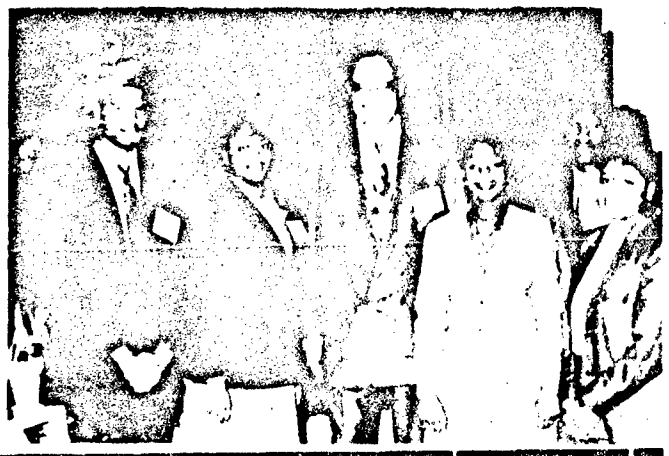
HYATT REGENCY CRYSTAL CITY, VIRGINIA

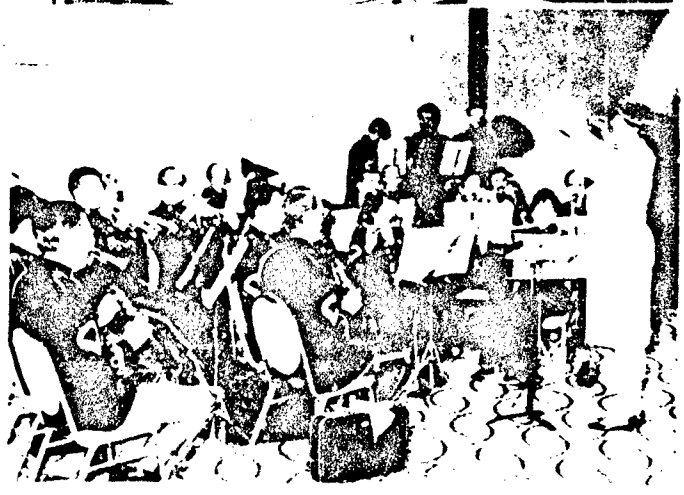
1992 Ada Conference

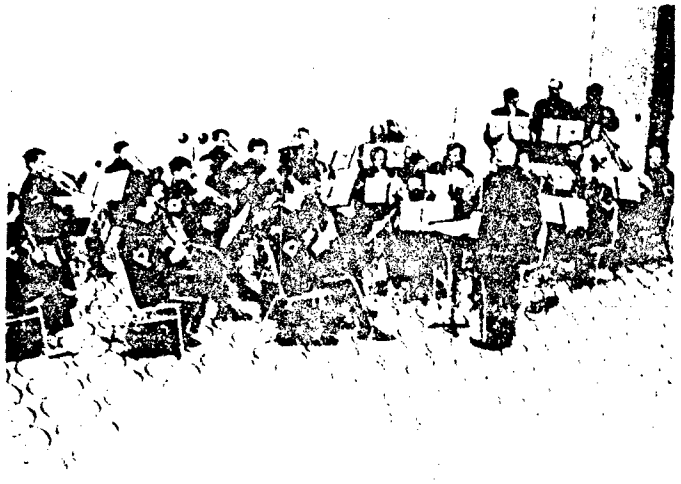


Ada Conference Highlights

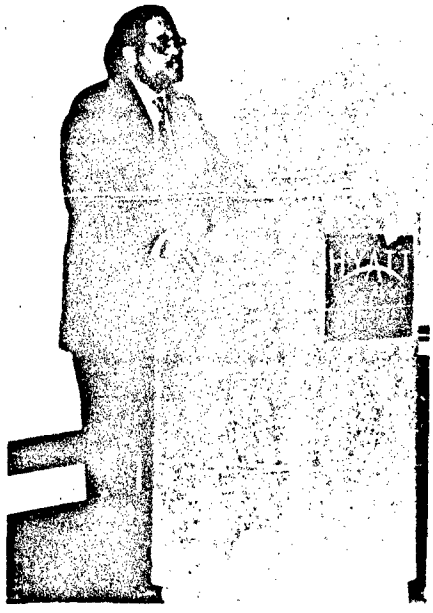








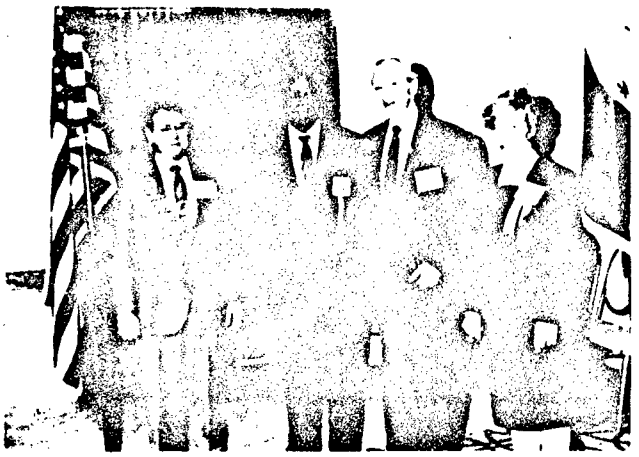


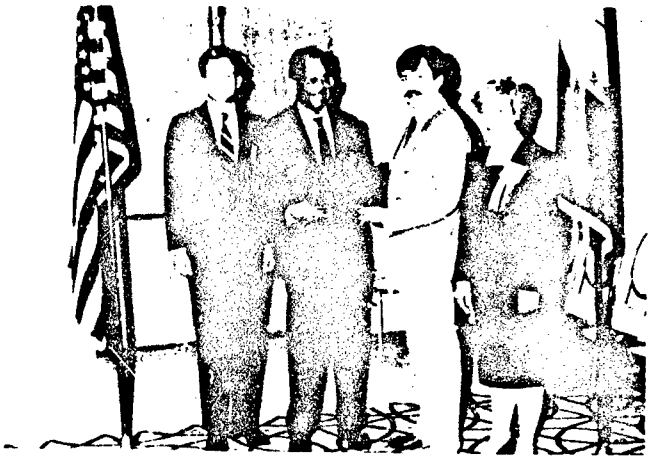


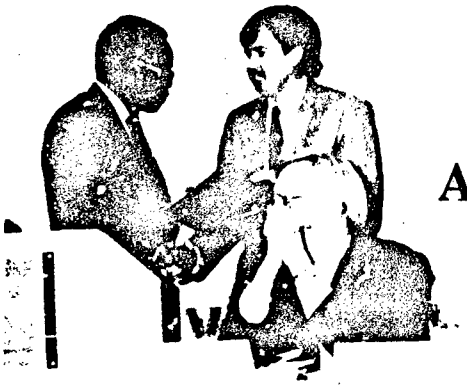

1992 Ada Conference Keynote Speakers









1992 Ada Conference

Awards and Thank You's

Table of Contents

Tuesday, March 16, 1993

Wednesday, March 17, 1993

Thursday, March 18, 1993

PROCEEDINGS
ELEVENTH ANNUAL NATIONAL CONFERENCE ON Ada TECHNOLOGY

Bound-Available at Fort Monmouth

2nd Annual National Conference on Ada Technology, 1984--(N/A)
3rd Annual National Conference on Ada Technology, 1985--$10.00
4th Annual National Conference on Ada Technology, 1986--(N/A)
5th Annual National Conference on Ada Technology, 1987--(N/A)
6th Annual National Conference on Ada Technology, 1988--$20.00
7th Annual National Conference on Ada Technology, 1989--$20.00
8th Annual National Conference on Ada Technology, 1990--$25.00
9th Annual National Conference on Ada Technology, 1991--$25.00
10th Annual National Conference on Ada Technology 1992--$25.00
11th Annual National Conference on Ada Technology 1993--$25.00

Extra Copies: 1-3 $25 each; next 7 $20 each; 11 & more $15 each.

Make check or bank draft payable in U.S. dollars to ANCOST and forward requests to:

Annual National Conference on Ada Technology
U.S. Army Communications-Electronics Command
ATTN: AMSEL-RD-SE-CRM (Ms. Kay Trezza)
Fort Monmouth, NJ 07703-5000

Telephone inquiries may be directed to Ms. Kay Trezza at 908/532-1898.

Photocopies--Available at Department of Commerce. Information on prices and shipping charges should be requested from:

U.S. Department of Commerce
National Technical Information Service
Springfield, VA 22151
USA

Include title, year, and AD number

2nd Annual National Conference on Ada Technology, 1984-AD A142403
3rd Annual National Conference on Ada Technology, 1985-AD A164338
4th Annual National Conference on Ada Technology, 1986-AD A167802
5th Annual National Conference on Ada Technology, 1987-AD A178690
6th Annual National Conference on Ada Technology, 1988-AD A190936
7th Annual National Conference on Ada Technology, 1989-AD A217979
8th Annual National Conference on Ada Technology, 1990-AD A219777
9th Annual National Conference on Ada Technology, 1990-AD A233469
10th Annual National Conference on Ada Technology, 1990-AD A248007

Keynote Speaker

Dr. Barry Horowitz
President & CEO
The MITRE Corporation

Dr. Barry M Horowitz is MITRE's president and chief executive officer. He served from 1987 through 1990 as executive vice president and chief operating officer, responsible for the general management and direction of the company's overall technical, financial, and administrative activities. Earlier, he was group vice president and general manager of the company's operations in Bedford, MA, which are primarily in support of the Air Force. Dr. Horowitz became a member of MITRE's Board of Trustees in February 1989.

Dr. Horowitz joined MITRE in 1969 at the company's Washington Center located in McLean, VA, and held a succession of positions from technical staff to department head. Most of his effort was devoted to air traffic control, in support of the FAA. During the period from 1969 to 1979, he became a recognized leader in the aviation community on the design of new collision avoidance systems. He led MITRE efforts, which initiated the technical approach for airborne collision avoidance systems, currently being established as a national standard. He also led MITRE efforts that have developed advanced enroute ATC automation concepts and designs, and initiated an activity to build a real-time simulation facility to evaluate new designs and concepts for the FAA. This effort resulted in what currently is a major FAA resource for designing advanced automation systems. He also played a major role in the analysis and regulation process that resulted in new government standards for independent IFR approaches to parallel runways.

During the period from 1979 to the present at MITRE, Dr. Horowitz moved to the company's headquarters in Bedford, MA, and rose in rank from Director of Special Studies to his present position. In addition to his management role, he has been a strong personal contributor to a wide variety of initiatives in the area of strategic command and control. He is a national authority on techniques for managing engineering programs.

Dr. Horowitz received a BS in Electrical Engineering from City College of New York in 1965, an MS in Electrical Engineering from New York University in 1969.

Acquisition Panel

LTG Alonzo E. Short, Jr.
Director, Defense Information Systems Agency
Manager, National Communications System

Lieutenant General Alonzo E. Short, Jr. was born in Greenville, NC on 27 January 1939. He grew up in Portsmouth, VA, where he attended I.C. Norcom High School. General Short holds a BS in education from Virginia State College and an MS in business management from New York Institute of Technology, Long Island, NY. He also holds honorary doctorates from Virginia State University and C.H. Mason University, San Diego, CA. His military education includes completion of the Signal Officer Basic and Advanced Courses, the Armed Forces Staff College, the Communications-Electronics Systems Engineer Course, and the Army War College.

Since entering the Army in June of 1962, General Short has held a variety of assignments with progressively increasing responsibility throughout his career.

In 1975, General Short was assigned as a staff officer in the Defense Communications Agency (now the Defense Information Systems Agency). Following that assignment, General Short was a battalion commander in the 101st Airborne (Air Assault) Division at Fort Campbell, KY.

In 1979, he began a tour as a staff planner with the Army Communications Command at Fort Huachuca. He then served as Commander of the 3rd Signal Brigade at Fort Hood, TX.

The General was then assigned as the Deputy Commander of the Army Electronics Research and Development Command (ERADCOM), Adelphi, MD, from July 1984 to October 1984.

General Short was promoted to Brigadier General concurrent with assuming command of the Information Systems Management Activity (ISMA) while at the same time taking over as Program Manager for Army Information Systems, all at Fort Monmouth, NJ. In July, 1986, he became Deputy Commander of the Informations Systems Engineering Command (ISEC), and Commander in September 1987.

General Short was promoted to Major General when he became Information Systems Command Deputy Commander on 7 September 1988. General Short then assumed command of the Information Systems Command in June 1990 and was promoted to Lieutenant General at the same time.

In August 1991, General Short became Director, Defense Information Systems Agency/Manager, National Communications System.

Luncheon Speaker

Donald E. Mullikin
Deputy Program Manager for Advanced Automation, AAP-2A
Federal Aviation Administration

Donald Mullikin is Deputy Program Manager for Advanced Automation, AAP-2A. Over a 20 year period, Dr. Mullikin held key technical, program/project management, and senior management positions in the Department of Navy and the Defense Intelligence Agency. He joined the FAA in November of 1984 and served in several technical and management positions in the Program Engineering Service. He joined the Advanced Automation Program in June 1988.

Dr. Mullikin holds a BS in Electrical Engineering, an MS in Electrical Engineering and PhD. in Electrical Engineering.

Prior to his current position, Dr. Mullikin was the Assistant Manager/Manager, Advanced Automation Systems (AAS) Division, Automation Service, Advanced Automation Program, FAA, Washington, DC. From July 1987 to July 1988, he was Assistant Manager, Facilities Integration Division, Program Engineering Service, FAA, Washington DC. From November 1984 to July 1987, Technical Advisor to Director, Program Engineering and Maintenance Service, FAA, Washington, DC.

From April 1981 to November 1984, he was Assistant Deputy Director for Defense Intelligence Systems, Washington DC. From June 1973 to April 1984, he was Assistant Manager, C3 Project Office, Naval Electronics System, Washington DC. From November 1968 to June 1973, he was Systems Engineer, Command Support Division, ASW Systems Project Office, Washington, DC. From September 1966 to November 1968, he was Electronics Engineer, Sonar Project Office, Naval Ship Systems Command, Washington, DC. From June 1964 to September 1966, he was an Engineer Trainee, System Effectiveness Section, Naval Ship Engineering Center, Washington, DC. From September 1962 to February 1964, he was an Engineering Assistant, Data Processing Division, Department of Commerce Bureau of Census, Suitland, MD.

Keynote Speaker

LTG Peter A. Kind
Director of Information Systems for Command, Control, Communications and Computers, Office of the Secretary of the Army

Lieutenant General Peter A. Kind is a native of Wisconsin. Upon completion of studies at the University of Wisconsin in 1961, he was commissioned a Second Lieutenant and awarded a BS in Economics. He also holds a MBA from Harvard University. His military education includes the Basic Officer Course at the Signal School, the Communications Officer Course offered at the US Marine Corps Amphibious Warfare School, the US Army Command and General Staff College and the US Army War College.

He was assigned to the 97th Signal Battalion (Army), 10th Special Forces Group (Airborne) in Germany and as Signal Advisor to the 21st Infantry Division (Air Assault) in Vietnam.

Following duty as Assistant Division Signal Officer of the 82d Airborne Division and as Executive Officer and S2/S3 (Intelligence/Operations and Training) for the 82d Signal Battalion, Fort Bragg, NC, he served in the War Plans Division of the Office of the Deputy Chief of Staff for Operations and Plans, Headquarters, Department of the Army. He commanded the 1st Calvary Division's 13th Signal Battalion, Fort Hood, TX and studied at the Logistics Management Center's School of Management Science. General Kind then served as Chief of the Concepts and Studies Division, Directorate of Combat Developments at the Signal Center prior to Army War College attendance.

He then served as Commander of the 1st Signal Brigade with concurrent duty as the Assistant Chief of Staff, J6, US Forces in Korea and G-6, Eighth US Army; as Director of Combat Development, and as Deputy Commanding General and Assistant Commandant, US Army Signal Center and School, Fort Gordon, GA. He served as Deputy Controller of the NATO Integrated Communications System Central Operating Authority, NATO's equivalent to the US Defense Communications Systems, headquartered at SHAPE, Belgium, and then as Program Executive Officer, Command Control Systems, Fort Monmouth, NJ.

He then served as Commanding General, US Army Signal Center and Fort Gordon Commandant, US Army Signal School, Fort Gordon, GA and as Commanding General, US Army Information Systems Command, Fort Huachuca, AZ prior to his present assignment as Director of Information Systems for Command, Control, Communications and Computers, Office of the Secretary of the Army, Washington, DC.

General Kind has been awarded the Distinguished Service Medal, the Legion of Merit (with Oak Leaf Cluster), the Bronze Star Medal (with 2 Oak Leaf Clusters), the Meritorious Service Medal (with 2 Oak Leaf Clusters), Air Medals, Army Commendation Medal and Senior Parachutist Badge.

Keynote Speaker

Gerald W. Ebker
Vice President, IBM
CEO, IBM Federal Systems

Gerald W. Ebker is chairman and chief executive office of the IBM Federal Systems Company. The company was formed in March 1992 and is responsible for iBM's operations with the federal government. It is one of several IBM companies operating as independent business units. FSC is a major provider of custom systems integration solutions, services and product offerings to government customers.

Mr. Ebker joined IBM Federal Systems Division in 1963 as a programmer on the Apollo space program at IBM Houston. Until 1973, he held a number of management positions while assigned to the space program.

From 1973 until 1976, he was program manager of the IBM advanced control system project for automating oil refineries. In 1977, Mr. Ebker was named manager of software systems at IBM Manassas, and in 1979 became general manager of FSD's facility in Gaithersburg, MD.

In 1981, he was named FSD vice president, Defense and Space Systems, and in 1983 he became FSD vice president, Complex Systems. He was named president of the Federal Systems Division in January 1987 and became an IBM vice president in February 1988. He was named president of the Systems Integration Division in April 1988 and became president of the Federal Sector Division in April 1990.

Mr. Ebker was named to his present position in March 1992.

In June 1992, he was elected to serve a one-year term as chairman of the Armed Forces Communications and Electronics Association.

Mr. Ebker has a BS in Mathematics from Harding College and a MS in Mathematics from Kansas State University.

Luncheon Speaker

RADM Robert M. Moore
Supply Corps, United States Navy
Commander, Naval Information Systems Management Center

Rear Admiral Robert M. Moore is Commander, Naval Information Systems Management Center. His responsibilities include information resources (IR) planning and policy related to the Department's multibillion dollar information resources budget. Additionally, he oversees IR-related programs, insertion of new technology, and acquisition of information resources.

A native of San Antonio, TX, he received a BS from the University of Texas and a MBA from Harvard University.

Commissioned in 1961, his early tours included Supply Officer of the Destroyer HYMAN and instructor at the Navy Supply Corps School at Athens, GA. In 1964 he was selected for duty in the Naval Nuclear Propulsion Program and in 1966 was assigned as the program's contracting officer at a division of the General Electric Company at Schenectady, NY.

From 1971 to 1973 RADM Moore served as Director, Nuclear Equipment Support Division at the Navy's Ships Parts Control Center, Mechanicsburg, PA, and following this, served a second tour in the Naval Nuclear Propulsion Program where he was in charge of the headquarters procurement and new construction budget functions.

He served as Supply Officer of the Submarine Tender HOLLAND at Submarine Refit Site One, Holy Loch, Scotland, from 1979 through 1981. He then served as Assistant for Program and Budget, Attack Submarine Division, Office of the Deputy Chief of Naval Operations (Submarine Warfare).

In July 1983, RADM Moore became the Vice Commander, Navy Accounting and Finance Center, Washington, DC.

In July 1985, he assumed command of the Navy Fleet Material Support Office, Mechanicsburg, PA. As the Senior Executive of the Navy's largest Data Processing System Development Center, his duties included direct responsibility for three of the largest Data Processing projects ever undertaken in the Federal Sector.

His first tour as a flag officer was Competition Advocate General of the Navy. RADM Moore was the Assistant Commander for Inventory and Systems Integrity, Naval Supply Systems Command from July 1988 to June 1991. He was responsible for Navy Programs to upgrade and modernize the Navy Supply Systems which supports the fleet throughout the world and led the program to insure the accuracy and integrity of the Navy's multibillion dollar logistics support inventories.

RADM Moore's military decorations include the Legion of Merit (five awards) and the Meritorious Service Medal (two awards). He is also qualified in submarines.

Acquisition Panel

Moderator: LTG Alonzo E. Short, DISA

Panelists:

FAST ANALYTICAL SIMULATION OF MISSILE FLIGHTS

Yuh-jeng Lee
Computer Science Department
Naval Postgraduate School
Code CS/LE
Monterey CA 93943

John V. Waite
Pacific Missile Test Center
Code 1051
Point Mugu CA 93042

Abstract

We present an air-to-air missile flight simulation that has been designed and developed using the Ada programming language with the object oriented methodology. It was aimed at providing a test and evaluation method that is more understandable, modifiable, efficient, and reliable than earlier FORTRAN simulations. The principles of abstraction, information hiding, modularity, high cohesion, and low coupling were used to achieve these goals. The resulting simulation, a three degree-of-freedom model of guided air-to-air missile, is an accurate mapping of the problem space into software. The simulation is primarily intended to study missile kinematics.

INTRODUCTION

The ever increasing cost and complexity of modern weapon systems forces new demands on the test and evaluation (T&E) process. More extensive testing is required with fewer resources. This paper explores one aspect of the T&E process as it relates to air-to-air guided missiles.

In the early days of missile T&E (*circa* late 1940s), missile performance capability was determined solely through flight test, that is, actual missile launches. The realization that all the T&E data requirements could not be met with a limited number of launches led to captive-carry flight test, laboratory testing, and simulation to complement the missile launches. Today's data requirements have grown in response to the increased missile sophistication and mission complexity. It is not unusual for a single flight test to cost more than a million dollars. Due to the increased data requirements and increased cost of flight test, missile flight simulation is receiving more and more attention.

Missile Flight Simulation

There are three levels of missile flight simulation in terms of cost and complexity:

- Real-time hardware-in-the-loop (HIL) simulation integrates actual missile hardware with special test and instrumentation equipment in a laboratory environment. The simulation software typically runs on a high-speed special purpose computer that drives the test equipment and missile hardware. The real-time HIL simulation requires a major development effort of approximately thirty-five to forty man years and costs from five to ten million dollars[1].
- The second level of simulation is the all digital six-degree-of-freedom (6-DOF) missile flight simulation. Six-degree-of-freedom indicates that the simulation computes forces and moments for all three axes. The 6-DOF simulation incorporates sophisticated models for various missile subsystems and runs on a mainframe class computer. The 6-DOF runs many times slower than real-time. For example, an actual missile flight that might take thirty seconds to complete in real-time might take eighteen hours to run to completion using a 6-DOF simulation. The 6-DOF simulation requires a development effort of four to six man years.
- The third level of simulation, the main focus of this paper, is the fast analytical simulation (FAS). Simulations of this class are a rapid and inexpensive tool allowing missile systems analysts to study overall missile response or capability expeditiously. The FAS is a three degree-of-freedom (3-DOF) simulation, usually the forces are computed for all three

axes (the moments are ignored) and a three dimensional space is represented. Alternately, a 3-DOF might represent a planar two dimensional view where forces acting on two axes are computed while moments are computed about the remaining axis. The FAS is intended to be easily accessible via personal computers to provide results in a timely fashion. A user enters initial conditions and results are presented within a few minutes.

Current Practices: Problems and Limitations

Current simulations are usually developed by physicists or aerospace engineers (who usually have little or no training and knowledge in modern software engineering principles), using the FORTRAN programming language. Their main goal is "just to get something up and running". The resulting simulations are almost always poorly structured and violate most commonly accepted computer programming principles. Typical characteristics of these simulations are:

- The simulations are monolithic pieces of code using many GOTO statements;
- most variables are treated as global;
- Common data areas are used for communication between subroutines;
- cryptic variable names are used (FORTRAN variable names are limited to six characters);
- the simulations are limited to very simple data structures (multi-dimensional arrays are usually the most sophisticated data structures found);
- programming through side is common; and
- the simulations have little or no comments or formal documentation.

The new analyst will usually require at least six months to gain a basic understanding of how the simulation works, even if he or she has an excellent understanding of missile systems. An understanding of the simulation is critical if results are to be interpreted correctly. It is not uncommon for the original developers of a simulation to move on to other jobs, leaving the analysts responsible for maintaining and modifying the simulation. Changes in the production missile's software or hardware, regularly occurring events, must be accurately reflected in the simulation code. Changes or patches introduced to the simulation code invariably make the code more obscure and, more often than not, produce undesirable side affects or bugs. Debugging these types of problems is incredibly time consuming and difficult.

After numerous patches have been applied the simulation software becomes unreliable and inefficient. Wildly different results are obtained for slightly different initial conditions. The real-time HIL simulations no longer run in real-time. The 6-DOF simulations may take days to solve a problem and the 3-DOF FAS simulations take hours - what once required hours and minutes respectively. Disk and main-memory capacity become issues. What was once a tool enabling scientists and engineers to analyze complex systems has become an unwieldy demanding burden of questionable value.

Consequently, they are difficult to understand and modify, and inevitably become inefficient and unreliable.

Motivation and Goals

Given the current situation, what is needed is a method that more closely represents the problem space, allowing simulations to be developed that are easy to understand and modify.

The major purpose of this project is to explore the use of object oriented techniques using the Ada programming language, in conjunction with contemporary software engineering principles, to implement a missile flight simulation. This simulation should be easily understood in a reasonable amount of time and readily accommodate change. Additional goals include producing code that is efficient and reliable.

The Approach

We have designed and developed a fast analytical simulation (FAS) program, written in Ada, which is aimed at providing a reliable and inexpensive tool that allows missile systems analysts to study overall missile response or capability expeditiously. The missile flight simulation models a subset of the missile systems (including the Autopilot, Airframe, and Guidance), the kinematics (consisting of the missile dynamics and the missile-target geometry), and the target. At the top level view, the simulation computes the forces acting on the missile (e.g., thrust, drag, and gravity) and from these forces derives accelerations to compute the missile's spatial trajectory from launch to target intercept.

By adhering to object-oriented design principles (including abstraction, inheritance, information hiding, modularity, high cohesion and low coupling), the result is an accurate mapping of the problem space into software. Since the mapping preserves the real world view of the problem, we believe that our simulation program is more understandable, modifiable, efficient, and dependable than earlier FORTRAN simulations.

OBJECT-ORIENTED TECHNIQUES WITH ADA

The goal of object oriented techniques is to produce software that is understandable, easy to modify, effi-

cient, reliable, and reusable. Each of these characteristics is elaborated below:

Understandability: This is critical to the management of complex software systems. It is, without a doubt, the most important factor of a simulation to the analyst responsible for maintaining the simulation. The software solution (that is, the simulation) should be an accurate model of the real world problem. Software can be thought of as being understandable on both a micro and macro level. Code at the micro level should have a style that is very readable. At the macro level data structures and algorithms should be able to be identified as mapping from the real world problem space. Understandability also tends to be tied to the programming language used and its richness of expression.

Modifiability: Well designed software should readily permit change. Modification is usually required due to a change in requirements or to correct to an error. Changes in missile simulation code are required to explore new concepts, or as a result of missile hardware or software upgrades. Many changes are not planned. Ideally, changes should not alter the fundamental architecture of the software solution.

Efficiency: Efficiency is the optimal use of two fundamental computer resources - storage space and execution time. Both of these resources are dependent underlying hardware, yet both resources are equally dependent on the software. An efficient missile simulation should provide better user response and more functionality than an inefficient simulation.

Reliability: The goal of reliability is to prevent failure, and to some extent, recover from failure in a graceful manner. Failure in a missile simulation might be defined as anything from a program that crashes to a program that produces results that do not agree with flight test data or produces inconsistent results. A reliable missile simulation will provide results that are consistent with real world experiences and give meaningful indications when potential problems might arise (e.g., limits exceeded or incorrect user input).

Reusability: The goal of reusability is to provide software components to build software much the same way hardware engineers build circuits from standard off-the-shelf components. The development of software systems can be dramatically reduced by using software components that have already been debugged and tested. These components can form libraries of commonly used objects. Systems may be constructed from these libraries. These systems then may be added to the library.

Object-Oriented Principles

Through abstraction, information hiding and modularity, object oriented techniques encapsulates data and procedural abstractions to form objects. Objects modularize both information and processing, rather than processing alone. Object oriented techniques establish a mechanism for (1) a representation of data structures, (2) the specification of process, and (3) the invocation procedure. An object is an element of the real world mapped into the software domain. The object consists of operations which act on data structures in response to messages sent to that object from other objects. The operations and data structures are hidden, that is the implementation details are unknown to the user of the object. The interface to the object is the only portion visible to the user. The interface is a set of well defined messages that specify what operation on the object is desired. Object oriented techniques can aid sound software engineering principles. These principles include abstraction, information hiding, modularity, loose coupling, and strong cohesion[2].

Abstraction. Many of the problems found with the missile flight simulations are due to their complexity. Abstraction is a powerful concept that helps one deal with complexity. Abstraction concentrates on the essential aspects of a problem, while omitting the details. There may be many levels of abstraction constructed when solving a problem. At the top level of a missile simulation, abstraction would reveal the essential entities - the missile, target, and environment. Moving to the next lower level of abstraction within the missile, this level might be thought of as being composed of various subsystems, such as the seeker, the guidance section, the autopilot, and the airframe. Moving to the next lower level of abstraction, arbitrarily choosing the missile seeker for example, would reveal the data structures and procedures used to model the seeker. Only the lower levels of abstraction expose the specific details of a solution.

Information Hiding. Information hiding conceals the implementation details of a solution that should not affect other parts of a system. Through information hiding only the essential aspects of a solution are visible, while the implementation details or "how" of a solution are hidden. Hiding low level design decisions prevents the higher levels of abstraction from being dependent on implementation details. This approach aids abstraction and increase the modifiability of the solution.

Modularity. The importance of modularity in software design has been recognized for some time. According to Myers[3], "Modularity is the single attribute

of software that allows a program to be intellectually manageable." In monolithic software, such as the missile simulations, the number of control paths, number of variables and the overall complexity make understanding difficult. Ideally, software is decomposed into modules along logically and functionally independent lines. Modularity supports our notion of abstraction. High-level modules specify "what" is to be done. Low-level modules specify "how" that action is to take place.

Cohesion and Coupling. Modules in software systems can be thought of as having two important characteristics, cohesion and coupling. Cohesion attempts to characterize to what degree a module performs a single function or serves a single purpose. A highly cohesive module would be one in which the module performs a single task that requires little or no interaction with other modules in a program. A module exhibiting low cohesion would perform many different functions and interact with a large number of other modules. Modules that are highly cohesive are easier to understand and are more amenable to change than modules exhibiting low cohesion.

Coupling is measure of interconnection among modules in a program. Modules with high coupling have a complex interfaces and make use of data or control information found in other modules. Modules with low coupling have relatively simple interfaces and make use of only the data or control information presented by the interfaces of other modules. Changes made to modules with low coupling are less likely to cause unwanted effects in other modules, that is the ripple effect is minimized. Like modules that are highly cohesive, modules with low coupling are easier to understand and modify.

Inheritance. Inheritance is an object oriented concept that permits the organization, building and reuse of software[4]. In a limited view of this concept, new objects may be defined to inherit the capability and functionality of other previously defined objects. The new objects may extend the capability and functionality of the original object by adding new capabilities and functionalities. Conversely the new object may be defined to eliminate or limit certain capabilities of the original object. Once an object has been developed, it may be reused with minimal effort through inheritance, reducing development time.

Object Oriented Methodology with Ada

Object oriented techniques build on sound software engineering principles to encapsulate data and procedures into objects. They capture the real world problem space, map well into the Ada programming language.

Ada Packages. The object oriented philosophy maps well into the Ada programming language. Ada has a wide set of constructs for providing primitive objects and operations. These constructs serve to build the implementation level of the objects. Ada's packaging concept is conceptually similar to objects and provides the means to encapsulate objects. According to Booch[5], "A package is a collection of computational resources, which may encapsulate data types, data objects, subprograms, tasks or even other packages." An Ada package consists of a specification and a body[6]. The specification identifies the information that is visible to the user of that package. The package body contains the implementation details of the package which should (and can) remain physically and logically hidden from the user. The specification and body may be compiled separately to enforce the separation of the specification or interface from the body with its implementation details. The specification can serve to define the messages associated with an object. The object responds in the appropriate manner to these messages. These messages might map to function or procedure calls and their input or output variables.

Ada packages can be used to provide reusable software components. Packages of commonly required objects can form libraries where they may be withdrawn and reused. Ada's generic unit feature supports, in a limited way, the object oriented principle of inheritance. A generic package serves as a template for an object[7]. The generic object can then be instantiated with all the features of the generic object, along with any additional features required of that particular instantiation. For example, a generic stack or list object might be instantiated for each occurrence of a different data type, along with the additional capabilities that make sense for that particular data type.

Methodology. We used an object-oriented development technique similar to that advocated by Booch[5] and first proposed by Abbott[8]. The development process involves five steps:

1. First, identify the objects and their characteristics or attributes as they exist in the problem space. Often a concise problem statement is useful in identifying objects. The nouns of the problem statement serve to identify potential objects.
2. The second step is to identify the operations that characterize the behavior of the objects identified in the first step. These should be meaningful operations that can be performed on the object. Verbs associated with an object noun in the problem statement can aid in the identification of meaningful

operations. During this step time and space constraints are formed to define the dynamic behavior of the objects. The scope and ordering of operations might be defined for example.

3. The third step is to establish the visibility of the objects with relation to one another. This step attempts to specify what objects "see", and what are "seen" by a given object. This serves to map the problem space into the objects.
4. The fourth step is to define the interfaces to the objects. To do this an object specification is produced which "forms the boundary between the outside view and the inside view of an object." This maps directly into the Ada package specification construct.
5. The final step is to implement each object by designing suitable data structures and algorithms and to implement the corresponding interface from the fourth step. Also at this step it is important to remain aware of the software engineering principles of modularity, high cohesion and low coupling. Note that this whole process can be recursive, that is, an object might further be decomposed into subordinate objects.

The key point of this method is the accurate mapping of the problem space into software. This mapping preserves the real world view of the problem, and if done properly, tends to produce code that is easily understood. Object oriented techniques also lend themselves well to the software engineering principles discussed earlier. Through object oriented techniques and sound software engineering principles, our goals of producing a missile flight simulation that is easy to understand, easy to modify, efficient and reliable can be realized.

THE PROBLEM SPACE

An air-to-air guided missile is designed to be carried on an aircraft and launched at an airborne target. After launching the missile guides, using its sensors, on the target to intercept. Air-to-air missile sensors may be radar, infrared or a combination of both types. Once the missile detects the target, it tracks and guides on the target by generating steering commands that will set a course to intercept the target. The missile flight simulation attempts to represent or model the missile and its environment.

The missile flight simulation models a subset of the missile systems, the kinematics, and the target. At the top level view, the simulation computes the forces acting on the missile (e.g., thrust, drag, and gravity) and from these forces derives accelerations to compute the missile's spatial trajectory from launch to target intercept. Figure 1 is a top-level block diagram of a missile

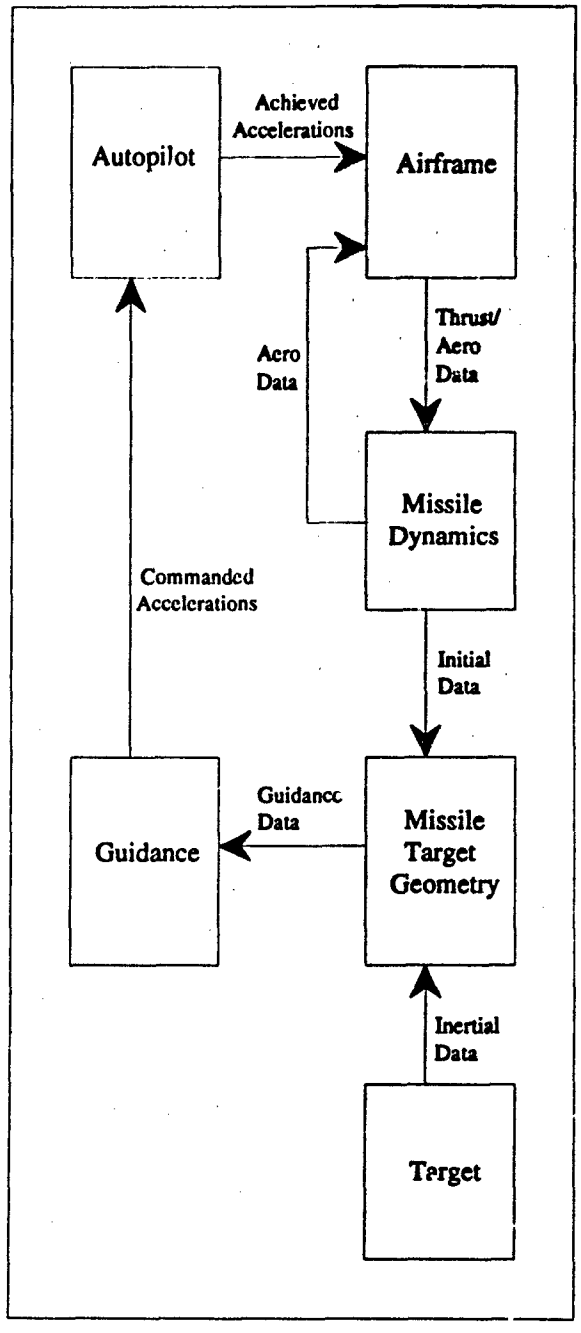


Figure 1: Missile Flight Simulation

flight simulation indicating the relationships of the various models. The missile subsystems are represented by the Autopilot, Airframe, and Guidance blocks. The blocks labeled Missile Dynamics and Missile-Target Geometry compute the kinematics. The Target block here represents a single target, but in most simulations more than one target is modeled. The missile airframe model, given its achieved accelerations, computes the forces acting on it for use in the kinematics model. The kinematics model calculates missile dynamics and the missile-target geometry, which is passed to the guidance model Acceleration commands, which will enable the missile to intercept the target, are computed by the guidance model and provided to the autopilot. The autopilot responds with the achieved accelerations, which are passed to the airframe model.

The Airframe

The missile airframe actuates or deflects the control surfaces which steer the missile. The missile is modeled as a rigid body and, as such, body and control surface bendings are not represented. The airframe is represented in terms of a reference axes system. Three force equations describe the forces experienced by the missile along each axis. There is one force equation for each axis as follows:

$$\Sigma F_x = m * a_x$$
$$\Sigma F_y = m * a_y$$
$$\Sigma F_z = m * a_z$$

These represent Newton's classic relation that force is the product of mass and acceleration. Here we resolve the forces into components along each missile axis.

These force equations describe the dynamics of the airframe. Aerodynamics is the science applied to predicting these forces. These forces are expanded in terms of aerodynamic parameters and coefficients[9]. For example, the x-axis equation becomes:

$$\Sigma F_x = m * a_x = F_p + (Cd_\alpha * \alpha q * S) + (Cd_\delta * \delta * q * S)$$

The first term in the above equation is the propulsion force. The second term is the product of the drag coefficient for a given angle of attack (Cd_α), the angle of attack (α), and the missile reference area (S). The third term is the product of the drag coefficient for a given control surface deflection (Cd_δ), the control surface deflection (δ), the aerodynamic pressure (q), and the missile reference area (S). The aerodynamic coefficients are a function of angle of attack, control surface deflection, roll angle and mach number. The aerodynamic parameters and forces are provided to the airframe model by the missile dynamics model, and the autopilot model provides the commanded accelerations as input. The airframe model computes the forces it is "experiencing" and sends these values to the kinematics model.

Kinematics

The kinematics model serves two functions: to compute missile dynamics and to compute the missile-target geometry. Inputs to the missile dynamics function are the airframe forces computed in the airframe model. From these values, and from initial conditions, the dynamics model derives acceleration, velocity, position data, and flight-path variables. Angles, angular rates, and accelerations represent the inertial quantities. These inertial quantities simulate the inertial sensor measurements the missile would experience.

Aerodynamic parameters, such as mach and velocity, are fed back to the airframe model. The derived acceleration, velocity, and position variables are sent to the missile-target geometry model. The kinematics model also transforms data between the two reference coordinate systems, that is, between the airframe reference system known as the body coordinate system (x, y, z) and the autopilot reference system known as the inertial coordinate system (X_a, Y_a, Z_a).

The primary purpose of the missile-target geometry portion of the kinematics model is to compute the missile-target engagement geometry parameters. These values are sent to the missile guidance model to steer the missile to the target. Inputs to the missile-target geometry model are missile acceleration, velocity, and position data from the missile dynamics model, and target acceleration, velocity, and position from the target model. These inputs are used to compute range rate (closing velocity), missile to target range, line-of-sight (LOS) rate, LOS, and time of flight. The simulation is terminated on range or time constraints determined by this model. The computed information is sent to the missile guidance model.

Missile Guidance

The guidance model represents the missile guidance law which determines what trajectory will cause the missile to intercept the target. Missile guidance can be classified by the type of sensor is used to provide target information. Common sensors are RF (radar), or infrared (IR). A missile may use a combination of RF and IR seekers. The actual missile guidance section is very complex and sophisticated, hence, extremely difficult to model. Most simple simulations assume a perfect guidance section that uses a modified proportional guidance law.

An important parameter in guidance is the line-of-sight (LOS). The line-of-sight is the direction the missile "looks" in order to "see" the target. This is an imaginary line from the missile's seeker to the centroid of the target. It has been proven that, given constant target and missile velocities, if the LOS angle between the target and the missile remains constant an optimum trajectory will be achieved, result-

ing in a minimum miss distance[1]. If the LOS angle is to remain constant then the LOS rate must be zero. The LOS rate is computed in the guidance section and multiplied by the navigation ratio N and sent to the autopilot as commanded accelerations proportional to the LOS rate; hence, the term proportional guidance.

The commanded accelerations will change the missile velocity relative to the target velocity, driving the LOS rate to zero. The response of the guidance system is determined by the value chosen for the navigation ratio N. Most modern missiles improve upon the pure proportional guidance law by using the target related information available through improved sensors and increased on board computing power. Target related parameters used in addition to LOS include target range, velocity, acceleration and time to intercept. Given guidance data, the guidance model computes the commanded accelerations required to intercept the target.

The Autopilot

The autopilot functions to give the missile stable and controlled flight. The autopilot has its own axes reference system (X_a, Y_a, Z_a). The airframe motions about the autopilot axes are controlled by the autopilot. Motions about the Ya and Xa axes determine missile direction. The autopilots for these axes are termed the pitch and yaw autopilots, respectively.

The autopilot receives commanded accelerations as input and responds with achieved accelerations as output. The achieved accelerations are based on the characteristics of the autopilot and other missile subsystems.

Target

The target model represents a simple maneuvering target. Inputs are positional, heading, velocity, and type of maneuver initial conditions. Inertial data are derived from this data and output to the missile-target geometry model. More sophisticated target models might include multiple targets and targets capable of complex maneuvers.

The Atmosphere

The Earth's atmosphere is a dynamically changing system, within which the missile must operate. The pressure, density and temperature of the atmosphere depend on altitude, location on the globe, the time of day and the season. In order to have a common reference atmosphere, a standard atmosphere has been defined by the U.S. Air Force[9]. The standard atmosphere gives mean values of pressure, density, and temperature as a function of altitude. Most missile flight simulations model the standard atmosphere.

THE SIMULATION

The models described in the previous section become the Ada objects in the object oriented approach. The relations between the models are represented by messages between the objects. These messages can request actions of objects or be in response to message requests for action. Objects may be constructed from other objects. For example, a missile is composed of objects that represent subsystems. This approach results in a simulation that accurately maps the problem space to software, as illustrated below.

The Control Objects

At the highest level of abstraction, we wish to keep the simulation independent of the domain of applications. We might wish to simulate a power plant or missile – our upper most level should not reflect what particular application we are using. Objects are necessary to control or manage the simulation. By controlling or managing the simulation we mean things like getting user input, initialization, starting and stopping the simulation, and presenting data. The objects necessary for these operations are identified as the EXECUTIVE, APPLICATION, and USER_INTERFACE. Each is briefly described below:

- The EXECUTIVE has no knowledge of what type of simulation is running, it just sends a message to APPLICATION to initialize the system and a message to the USER_INTERFACE to turn over control of the simulation to the user.
- The APPLICATION object has the knowledge of the specific details of the application in terms of what objects exist and their interfaces. In our simulation, this object is responsible for initializing the system, starting and manipulating the simulation operations, and logging data. It also sends messages to the MISSILE and TARGET objects, requiring them to compute the mathematical derivatives that characterize them.
- The USER_INTERFACE object is required for user input and output. It allows the user to the user to control the simulation through keyboard input, along with presenting run-time displays and simulation status information to the user.

The System Objects

The system objects includes the missile, launcher, and target objects. The launcher object provides the missile with launch aircraft information and targets provides target information.

One of our major software engineering goals, understandability, is achieved by implementing and discussing the core of the simulation in terms of modular objects. This approach also serves to accurately map

the real-world problem space into the objects that form the software solution.

The Missile. It includes the following subsystem objects: AIRFRAME, AUTOPILOT, RF_SEEKER, IR_SEEKER, and GUIDANCE. The AIRFRAME contains further subsystems such as the AERO and THRUST. The missile subsystems serve as a good example of abstraction, modularity, low coupling and high cohesion.

The MISSILE object sends messages to establish the missile's physical characteristics (such as mass, drag, thrust, and initial phase of flight), the launch type, number of targets, number of stand off jammers (SOJs), electronic counter measures (ECM) power,and geometric initial conditions (such as ranges and heading angle).

The AIRFRAME models the missile's aerodynamic characteristics and thrust characteristics. It initializes the propulsion phase and various physical constants, computes the missile's coefficient of drag and angle-of-attack, and provides the thrust force and propulsion phase.

The AUTOPILOT accepts commanded accelerations and returns achieved accelerations dependent on the body responses of the missile.

The function of GUIDANCE is to guide the missile to the target. Simply stated, given missile and target position and velocity information, GUIDANCE determines the required acceleration commands for the missile to intercept the target.

The missile uses its RF (radar) or IR (infrared) seeker to get information about the target. At longer ranges, the missile simply receives the RF energy reflected off the target from the launch aircraft's radar. At medium ranges, the missile's on-board radar activates to provide target information. At short ranges, the missile activates its IR seeker to acquire and track the target in the terminal phase of flight.

The Launcher. The LAUNCHER object represents the aircraft that carries and launches the missile. It provides launch aircraft information (such as velocity and position, and certain radar characteristics based on the launch aircraft type) for use by the missile. Other functions include providing the distance from the launch aircraft to the target and the launch aircraft's velocity if it is in the pursuit guidance mode.

The Target. The TARGETS object models the aircraft that the missile is to intercept. The Ada package TARGETS is made up of four targets. Two of these targets, target one and target two, are treated as the primary targets, and targets three and four are treated as stand off jammers (SOJs). The major task of TARGET is to calculate the targets' position, velocity and heading angle dependent on target maneuver. Appropriate conditions are checked resulting in the setting of flags and the times for the corresponding target maneuver. Then the build-up time must be considered. The build-up time is the time from the initiation of the maneuver until the desired number of g's is achieved. This models the real world condition that commanded maneuvers are not achieved instantaneously. The rate of change of the target's heading is then calculated along with the number of g's the target is experiencing. The current target heading angle is then compared with the final desired turn angle. The target velocity vector and mach are then computed. Finally, the second target's position is computed based on its geometric relation with the first target.

The Support Objects

Modular design and information hiding allow the simulation to be machine independent. Our simulation was developed and implemented on an IBM AT compatible machine. However, the simulation can be modified to run on other systems. To aid this process, all the machine dependent code is implemented (hidden) in the SYSTEM_SPECIFIC object. Most of this code is associated with the video display. By rewriting SYSTEM_SPECIFIC for the other hardware platforms, and keeping the original message names, the porting process should consist simply of recompilation of SYSTEM_SPECIFIC and a link of the simulation. Also by working at a lower system-specific level all screen displays are output in the most efficient manner providing very fast screen updates. This prevents the user from perceiving a delay as the screen is updated or the next menu is displayed (problems experienced in earlier FORTRAN simulations).

Other support objects include (1) INTEGRATION, which performs the numerical integration of the MISSILE and TARGET state variables, (2) MATH, which provides external messages that perform all the basic mathematical operations required by the simulation, and (3) REAL_MATRIX, which is the generic object MATRIX_AND_VECTOR instantiated for the real data type, and provides a number of messages for operations on matrices and vectors because many of the quantities encountered in the simulation, such as forces, are best expressed in terms of vectors or arrays.

Object Messages and Implementation

The Ada with clause allows a package (or object) to access or view another package's specification. Package specifications define the interface to the package in terms of data structures, function calls and procedure calls available to the users of the package. In our

object oriented view, package specifications define the external messages that an object can respond to by eliciting some type of action or providing the sender with information. Internal messages are the functions and procedures that are in the body and not in the package specification, and therefore are for the exclusive use of that object.

A detail account of the actual FAS code and simulation scenarios can be found in a report by Waite[10].

CONCLUSIONS

We have explored using object oriented techniques and software engineering principles in conjunction with the Ada programming language to develop a missile flight simulation. By using these techniques and principles the problem space is accurately mapped into software. This, along with the principles of abstraction, information hiding, modularity, loose coupling, and strong cohesion produced a simulation that is easily understood, modifiable, efficient, and reliable.

Although understandability can be very subjective, all of the missile analysts who reviewed the simulation agreed that the code is much more easily understood than previous FORTRAN versions. Modularity, high cohesion, and loose coupling permitted the simulation to be modified in easily. Modules were designed to serve a single purpose and to make use of only the data or control information presented by the interfaces of other modules. All the interfaces are well defined and are standard for that particular module. A good example is the abstraction of the missile airframe subsystem. By being modular and having a standard well defined interface, this subsystem evolved from a program stub to a fairly complex model with minimal programming effort. Also by having a standard well defined interface between objects or modules, a library of different models can be built to explore different missile and target configurations. The simulation is simply relinked with the desired module. This allows a number of different models to be built relatively quickly. These models then can be used for comparison studies.

Through abstraction, information hiding, and modularity a very efficient user interface was developed. The simulation has also proven itself to be highly reliable, producing consistent results that agree with missile system expert's predictions. The simulation has also proven to be quite robust, surviving the most mischievous users without crashing.

ACKNOWLEDGEMENT

We would like to thank Ted Finsold for his help in providing technical assistance for missile flight simulations. This research was supported by direct funding from the Naval Postgraduate School.

REFERENCES

1. Eichblatt, E.J., *Test and Evaluation of an Air-to-Air RF Guided Missile*, Pacific Missile Test Center, 1 July 1981.
2. Pressman, R.S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill Book Company, 1987.
3. Myers, G., *Composite Structured Design*, Van Nordstrand Inc., 1978.
4. Cox, B.J., *Object Oriented Programming: An Evolutionary Approach*, Addison-Wesley Publishing Co., 1987.
5. Booch, G., *Software Engineering with Ada*, The Benjamin/Cummings Publishing Company, Inc., 1986.
6. Department of Defense, "Reference manual for the Ada programming language", ANSI/MIL-STD-1815A, Government Printing Office, Washington, DC, January 1983.
7. Cohen, N.H., *Ada as a Second Language*, McGraw-Hill, Inc., 1986.
8. Abbot, R., "Program Design by Informal English Description", *Communications of the ACM*, 1983.
9. Anderson, J.D., *Introduction to Flight*, McGraw-Hill Inc., 1987.
10. Waite, J.V., "An Ada Object Oriented Missile Flight Simulation", M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1991.

THE AUTHORS

Yuh-jeng Lee received his Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign. He is currently an assistant professor at the Naval Postgraduate School. His research interests include knowledge-based and autonomous systems, intelligent tutoring/training systems, automated software construction, and computer simulations. Dr. Lee is a member of the ACM, IEEE Computer Society, and American Association for Artificial Intelligence.

John V. Waite received his Master of Science in Computer Science degree from the Naval Postgraduate School. He is currently with U.S. Navy's Pacific Missile Test Center. His research interests include software engineering, object-oriented programming, and computer simulations.

Ada Application Program Interfaces to X.400 Protocol Services

Charles A. Eldridge,
Sparta, Inc.
7926 Jones Branch Drive, Suite 900
McLean, VA 22102

Summary

This paper describes the design and development of software to implement Ada Application Program Interfaces to X.400[1] message handling services. This development makes it possible to use Ada as a building block of future mail enabled applications that send and receive electronic messages via X.400 message handling systems. The software architecture in which these APIs are to be used includes (1) the application program that creates, sends, receives, and processes electronic messages, and (2) the message handling services which are provided with a standardized API compatible with C language programming. We report the development process and discuss the long-range impacts and lessons learned from this development. The impacts include a significant reuse potential. The lessons concern the prospects for complete Ada developments versus API use as in this development.

Introduction

Ada Application Program Interfaces (APIs) to X.400 services provide the procedure interfaces and data definitions to enable application programs to send and receive X.400 electronic messages. The development of Ada APIs to X.400 services is motivated by two important mandates for US Federal Government computing and network communications – Ada language use in software development, and use of International Standards Organization / International Telegraph and Telephone Consultative Committee (ISO/CCITT) protocols for the exchange of data between computer systems. This latter mandate falls under the Government Open Systems Interconnection (OSI) Profile, GOSIP. Another motivation for Ada APIs to network services is the demand for distributed applications. Major ones include electronic mail, teleconferencing, and file transfers.

X.400 services are used for exchange of electronic messages, including text and other data formats via distributed message handling systems. These services operate much like postal systems, providing writer-to-reader services, using multiple store-and-forward points. Delivery can be to a user terminal, to a user's file store, to a facsimile device, or even as hardcopy through a cooperating physical delivery service.

To date, Ada has not had widespread use in conjunction with ISO/CCITT protocols. Where US Government policy dictates that Ada must be used as the development language and that ISO/CCITT protocols must be used, per the GOSIP, there is a serious deficiency. One aim of the work reported here is to remedy this deficiency.

The X.400 APIs described here are designed for developing "user agent" (UA) applications that interact with the X.400 message handling systems. User agents submit and retrieve messages during sessions with "message transfer agents" (MTAs). A set of Ada APIs will be especially useful to agencies that develop or acquire user agents whose development must be in Ada for reasons of mission criticality, required trust, etc.

The selected Ada APIs are the projection of the X.400 API Association's (XAPIA's) 1990 C language specifications into the Ada language. These Ada APIs use definitions of message system procedures and data objects (i.e., messages and message component objects) isomorphic to those of the XAPIA, in order to facilitate easy binding to software products that conform to the XAPIA specification. Therefore, this work establishes a major basis for software reuse, enabling users to develop message agents in Ada, that can interwork with COTS message handling components.

Architecture

Model of Operation

The X.400 protocol standards define a model for electronic mail, as illustrated by Figure 1. The model defines User Agents (UAs) and Message Transfer Agents (MTAs). UAs assist users in composing, submitting, and retrieving mail, and they allow users to interact with the Message Transfer System (MTS). The MTS is composed of one or more MTAs. An MTA accepts submitted messages from UAs for delivery to other UAs and delivers to UAs mail received from other MTAs.

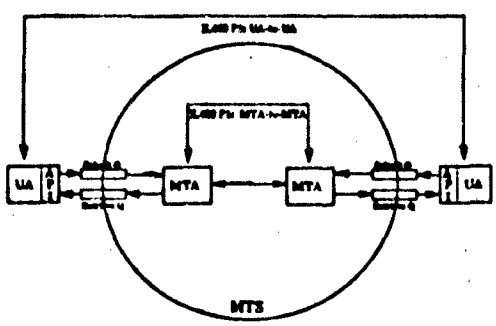


Figure 1.

Ada APIs in Message Transfer System

The Ada APIs provide access to X.400 Message Transfer System (MTS) services. These APIs are provided as a library of routines that are used by a user's application code (through the use of the Ada "with" directive). This combination of user's application code and API library may be thought of as an X.400 User Agent (UA), where the user code calls upon API services to submit and retrieve messages from the MTS. Figure 1 also depicts the relationship between the Ada APIs, the UA and the MTS components.

X.400 P2 is the protocol that prescribes the format of messages exchanged between two UAs. It includes the message originator, recipients, subject, text body, etc. X.400 P1 is the message format used between two MTAs; P1 is regarded as an envelope that includes the P2 messages.

To interact with an MTA, the UA must first open a session. A session is a set of message submission and retrieval transactions between a UA and an MTA. A UA submits a message to the MTA for delivery to another UA by placing the message on a submission queue ("SubmitQ" in Figure 1). Once the message has been placed on this queue, the message is under the control of the MTA. The MTA may relay the message to a remote MTA or may deliver the message to a local UA. A UA may submit any number of messages during a session.

Messages must be retrieved from the MTA by the receiving UA. When a message is received by the MTA, the MTA places the message on a retrieval queue ("RetrieveQ"). The receiving UA takes responsibility for a message, during a session, by removing it from the retrieval queue. The UA may also interact with the retrieval queue by querying for the number of currently available messages or by issuing a "wait" that causes the UA to suspend its processing until a message becomes available.

Once a UA has finished submitting and retrieving messages, the UA may close the session with the MTA. After the session is closed, the UA may no longer submit or retrieve messages. Messages that were in the submission queue when the session was closed are still sent by the MTA. Messages in the retrieval queue and delivery notices can be retrieved when a new session is opened.

To summarize, the Ada APIs include procedures for the following:

- open a session with the message handling service;
- close the session with the message service;
- submit a message for delivery;
- cancel a previously submitted deferred delivery message;
- start-retrieval: remove a message from the retrieval queue, accessed by the message sequence number;
- finish-retrieval: replace the message in the retrieval queue, or discard the message;
- size: obtain the number of messages in the retrieval queue;
- wait: await arrival of a message in the retrieval queue;

Relation to Standards

These Ada APIs were designed to meet two goals: (1) to make X.400 services accessible to Ada applications; (2) to provide specific APIs that can be

easily bound to XAPIA-compliant message handling software. By adhering to XAPIA standards and by using XAPIA message handling software (2), these APIs provide access to X.400 protocol services (1).

The Ada APIs and data objects are compatible with commercial message handling software systems that adhere to X.400 protocol standards. Therefore, applications using the Ada APIs can exchange electronic messages with a wide range of other applications, via X.400.

The Ada APIs can also be used with non-XAPIA-compliant message handling software, such as the University College London's PP[2]. In this case, additional procedures are needed to work between PP's APIs and the standard APIs. Figure 2 illustrates the two architectures for the use of the Ada APIs to X.400 services.

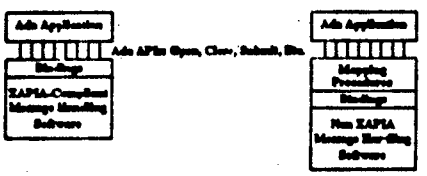


Figure 2. Two Architectures for X.400 Use

Development Approach

Ada Use To accomplish this development, we relied heavily on Ada's support for object oriented programming, including data abstractions and packages. We also relied on Ada's pragma interface for access to externally-provided X.400 protocol services (e.g., from Messenger 400 or from PP). Separate Ada packages contain the procedures for interacting with the message handling service, for defining message objects and for managing message objects.

This initial use of Ada to support message handling objects is simplified by the XAPIA's very clean separation of functions and data objects. The XAPIA object specifications are based upon an inheritance hierarchy, but we developed specific objects only for certain classes (messages, addresses, distribution lists, etc.).

Objects Messages are the top-level objects that are passed across the APIs. Messages contain other objects such as distribution lists, recipient addresses, originator addresses, etc. These objects in turn contain objects for the address components (Country, Organization, Surname, etc.). The XAPIA specifies this level of object definition; moreover, the message objects, so far, are passive and contain only attributes, not methods. The re-use potential of the object-oriented approach will be realized immediately during the development of mail-enabled applications.

Standard routines to manage objects can be used for both general and special objects as needed by mail enabled applications. For example, future mail enabled applications will likely include provisions for audio and video message parts, in both teleconferencing and non-real-time use. The object oriented programming techniques will be increasingly useful in future mail enabled applications.

Use of Services from External Software Early project constraints delayed acquisition of XAPIA compliant message handling software, Therefore, the first implementation of Ada APIs for X.400 services uses the public domain PP message handling software developed by the University College London, which in turn uses protocol services from ISODE (public domain C language implementations of ISO protocols). PP and ISODE are now supported by the ISODE Consortium.

Ideally, it will be very simple to bind services provided by commercial software developed in accordance the XAPIA specifications, to the Ada API software, via a small set of interface pragmas. Significantly, our use of PP, whose APIs are not XAPIA-compliant, also shows the use of Ada as the building block between APIs based on the XAPIA specifications and APIs particular to PP.

Test and Demonstration To exercise and test the APIs, we developed several simple application programs that manage objects and exchange them via the APIs. These were demonstrated over a local area network. An Ada application program creates and manages message objects, opens and closes sessions with a message handling service provider, submits message objects to it, retrieves message objects, and manages the contents of the message retrieval queue. The submitted and retrieved

messages were exchanged with another Ada application program, via PP message handling services, across the local area network.

Further Experience

The XAPIA-based approach is currently proving its re-use potential, in the development of a mail-enabled application program for teleconferencing. Teleconferencing is a much-needed and heavily used military command-center application. In its current version[3], it is not based on Ada, and it uses a command-line interface. Users receive messages from the floor, through a chairman. The mail-enabled teleconferencing uses the Ada APIs to X.400 services to exchange messages between conference participants, and it interacts with users via X window services.

Most teleconferencing applications rely on real-time protocols. However, the performance of prototype systems using PP allowed submitted messages to be retrieved within about 1 second by recipients. Therefore, X.400 services were chosen because they can provide for the other primary needs of a teleconferencing application, including basic message delivery, delivery to multiple recipients (i.e., the teleconference participants), delivery to conference records, etc.

We developed the application based upon the APIs described above, according to the model on the left hand side of Figure 2. We developed bindings as shown in that figure; code for these is simple and short. In other words, we were able to reuse the core work of the initial development, but did not reuse the mapping procedures shown in Figure 2.

We used vendor-provided bindings to XView window services to create and manage window objects. In particular, "button" objects cause specified functions to be executed when the button is pressed. For the teleconferencing application, these functions based on the Ada APIs to X.400 network services to perform message handling services in response to user actions. For example, the "Send" button activates a procedure that collects text from a text window, and then constructs a message to the teleconference that uses this text as the "Content", and finally invokes the "send" API.

In summary, the approach of using Ada APIs allowed us to develop an Ada application that makes use of up-to-date systems for both message handling and window management.

Software Engineering Implications

This work developed software resources intended specifically for re-use, with a very high leverage potential. This potential has been confirmed by early experience in actual re-use. This section will briefly discuss additional implications based on the development experience, and based upon projecting re-use in potentially large or critical applications.

Development Lessons

Complexity This development showed utility of development based upon standards, rather than more abstract requirements. The literal software requirements were quite bounded and not subject to change during the development. This fact constrained the growth of software complexity.

Using automated tools, we tracked the following complexity metrics during the development

- Cyclomatic Complexity: the complexity of the program execution graph;
- Total Operators and Unique Operators (Ada language operators, syntactic elements, etc.), and
- Total Operands and Unique Operands, .

Initial measurements were made on the Ada compiled Program Design Language (PDL) version. The complexity grew subsequently as various interface methods had to be developed, and as some functions were re-implemented inside more than one package. The Cyclomatic Complexity grew from 75 to 450; the total number of operators grew from 8,000 to 10,000; the total number of operands grew from 500 to 6,000. The number of lines of code grew from 600 to 2,600.

The particular solutions necessary to obtain a workable interface and functional translation between the XAPIA functions and those offered by PP were the greatest contributors to software complexity.

Fault Density We found that fault identification and repair in the development process worked reasonably well -- to enable the software to pass the Formal Qualification Tests. The density of faults from this basis (identified during development and review, and subsequently corrected prior to Formal Qualification Tests) was about 10 faults per 1,000 lines of code. This is less than might be expected from expert's experience[4] -- 20-25 errors per 1,000 lines of code.

Several advantages worked together against faults: stable requirements, stable design, skilled, intelligent development staff, and a test-and-demonstration covering a lot of execution paths. Had any of these been absent from the development, it is likely that significantly more faults would be experienced.

Implications for Future Use

The above-described development of the teleconferencing application can be a useful model for development of mission-critical command and control applications in Ada in which software verification is important. The APIs allow the application to use services of software developed elsewhere; they are essentially input-output services. The mission-critical functions, such as message validation, release subject to authorizations, authentication of senders, etc., can remain the responsibility of the Ada application. The use of Ada APIs can significantly reduce the effort needed to verify the application software as needed for military command and control applications.

The suggested model for future developments involves stable standards for the API to furnished services, such as message transfer, retrieval, and window management. The stable APIs would be a key for the widespread use of the services furnished via utility software and would free developers and verifiers to concentrate on functions essential to the application.

Conclusions

This work is significant because it is one of the first developments for Ada use of the International Standards Organization's Open Systems Interconnection (OSI) communication protocols. It is now a Federal Government mandate to acquire OSI protocols in new information systems.

This work demonstrated the use of two different X.400 message handling software systems by Ada application programs, plus the use of X-window services. This experience shows the power of the use of standard APIs for generic services. Develops can concentrate on the functions essential to the application and can use generic services of external or utility software by means of the APIs.

This work also demonstrated the potential of X.400 message handling systems to be used not only for message exchange, but for near-real-time teleconferencing as well. Despite the involvement of the disk file system, message transfers could be accomplished in approximately one second, sufficient for teleconferencing use.

Ada served as the primary building block throughout the developments described above:

1. it was the basis of mapping between the specified XAPIA APIs and the APIs for the public domain PP message handling software;
2. it was the basis of message object management utilities to be used with the message object containment hierarchy; and
3. it can be the basis of message processing for future mail enabled applications themselves.

Acknowledgements:

The author wishes to acknowledge the work of Triet Lu, of SAIC, Inc. and of Curt S. Kuhn, and Gordon E. Lee, SPARTA, Inc. for their contributions to the development of the Ada API software. This work was sponsored by the Defense Information Systems Agency, Defense System Support Agency. The author also wishes to thank Mr. George King of that agency for support, direction and encouragement.

References

[1]The International Telegraph and Telephone Consultative Committee, "Data Communication Networks Message Handling Systems. Recommendations X.400 - X.420", 1988.

[2]Kille, S., Implementing X.400 and X.400: The PP and QUIPU Systems. (Artech House, Boston, 1991).

[3]Worldwide Military Command and Control System (WWMCCS) Standard System Software. See Joint Data Systems Support Center Technical Memorandum TM 245-85, "WWMCCS Intercomputer Network (WIN) Telnet, Teleconference, and File Transfer Service User Guide.

[4]Gilb, T., Software Metrics (Winthrop Publishers, Inc., Cambridge, MA, 1977), p. 57.

Author Information

Dr. Charles A. Eldridge, Sparta, Inc., 7926 Jones Branch Drive, Suite 900, McLean, VA 22102, email: "eldridge@sparta.com". Dr. Eldridge is Principal System Analyst who earned a Ph. D. in 1976 at Yale University. He has worked as a software developer and researcher in scientific computing, computer communications and related areas for the past 16 years. He has also been employed by Cornell University, System Development Corp. and the MITRE Corporation.

ASTRODYNAMICS 101,
A CASE STUDY IN ADA OBJECT BASED PROGRAMMING

Roger V.Z. Kovacs
Graduate Student, University of Colorado at Colorado Springs

Abstract

A course of studies toward a Masters of Engineering, Space Operations was pursued A required prerequisite for this degree is a single programming class in Ada. This prerequisite was completed at the University of Denver, taking the graduate level Ada sequence. Shortly after completing the Ada curriculum, my employer arranged for me to attend an in-house Object Oriented Analysis class (using the Shlaer/Mellor[1] methodology). These classes, along with my work in the aerospace industry, emphasized the importance of Ada. A personal decision was made to complete all astrodynamic assignments in Ada and develop a reusable library of components. The programs were developed for the classes in Orbital Mechanics, Launch & Re-entry, and Astrodynamics. Although Ada is a required prerequisite, I believe that I was the only student out of the 30 or so students in my classes using Ada for the assignments. Other students used C, FORTRAN, and Pascal. While new to Ada, I was able to easily keep up with my colleagues, demonstrating that Ada and its various unique features are not difficult to learn, nor the cause of schedule delays. This experience provides some insight into problems existing in the Ada community. The case study serves as a starting point for others considering the use of Ada in their educational pursuit, as viewed by an engineering student outside of the computer science discipline. It can be used for discussions on the advanced topics of Ada, i.e., how to implement a project, good and bad methodologies, limitations, and allowing students to learn from other's attempts.

Education

For a large corporation to bid on some government contracts which require the use of Ada, the company has to identify its resources including the number of Ada programmers. The accepted definition of a trained Ada programmer is 60 hours of classroom training and 60 hours of hands on experience. This provides a very minimal education of Ada. Completion of this type of training provides the student with the basic understanding of the syntax and little more. I was fortunate to receive my Ada education in a fast paced environment of graduate computer science students. We approached the subjects of object philosophy, generics, and tasking in a detailed manner as implemented in Ada. The curriculum for this sequence of classes included the dining philosopher problem using a state machine and array of tasks for each of the objects. It became an eye opener as to the amount of processing that can be accomplished with so little code.

The most common student in an Ada class is an engineer with experience usually in FORTRAN, whose employer is considering the use of Ada. His/her engineering discipline may vary but for the most part it is applied outside of the computer science field. Typical disciplines are controls, navigation, guidance, or structures. These engineers leave their work assignments for a few weeks to receive Ada training. When they return to their jobs, and only when required, they begin to write in Adatran, Ada that looks exactly like the FORTRAN that they were previously using. The code uses "common blocks", no generics nor tasks, and is usually contained in one package.

The fault does not lie with these engineers. The system only provided time for learning a new syntax, not a new software engineering methodology. One of the philosophical goals of Ada is to improve the overall education in software engineering. This case study attempts to utilize some software engineering training to create an Ada software model based on more modern techniques and methodologies.

Data is Lacking

Many of the best features in Ada such as generics are underutilized due to the fact that standards and literature describing how to create a good generic are lacking. While this case study will not establish a standard, it may

contribute to the discussions and interests of other students to review, criticize and improve on the models developed in this case study.

The basic concepts of object oriented programming seem so abstract when attempting to tackle a problem in astrodynamics, but after many revisions, it ends up being very logical.

A Good Starting Point

This case study provides a complete working model that can be expanded, modified, corrected, improved, or discarded completely. However, it can provide that start necessary for discussion. In addition, it may provide some reusable components that an engineering student may find helpful in pursuit of their education. It should provide a challenge to educators and students to solve the problem in a better way.

Object Oriented Based Design/Code

Not everything in this world is black and white, nor should the standards on the best way to code. Software engineering, and engineering in general is a process or method used to solve a problem. There may exist an infinite number of possible solutions, but the engineer must decide on one solution which is a cost effective, safe, and practical solution. This usually results in a compromise. One piece of code cannot necessarily be the do-it-all for every application. Real time simulation engineers have different requirements than that of simulation engineers needing to work with very high fidelity models. The engineer is now tasked to not only solve the problem for today, but also for the future. To develop a reusable component, it must be practical, efficient and maintainable.

Practicality means ease of use. A complex generic routine using access types of records just to add two numbers would overwhelm anyone, even if it could be used by anyone for anything needed to be added. The component must first make sense. To make every object a limited private to enforce the highest levels of encapsulation and hiding may not provide a practical solution. Ease of use must be considered for the current and future users. Most engineers prefer to use assignment statements for their operations so that the code looks like the equation in the text book. For example, if the following equation is desired:

$$\bar{g} = \frac{1}{\mu_\bullet}\left(|\bar{v}|^2 - \frac{\mu_\bullet}{|\bar{r}|}\right)\bar{r} - (\bar{r} \bullet \bar{v})\bar{v}$$

The Ada would look like the code fragment in figure 1.

```
r := NORM (State_Data.Position);
v := NORM (State_Data.Velocity);
g_vec := (1.0/MU)
  * (((v**2 - MU/r) * State_Data.Position)
    -((State_Data.Postion*State_Data.Velocity)
      *State_Data.Velocity)) ;

where .Position and .Velocity are arrays, the
"*" is a dot product between vectors, and a
scalar * vector multiply as overloaded.  The "-"
is a subtraction between two vectors, resulting
in a vector.
```

Figure 1, Sample Ada Equation

Limited privates, since they exclude assignment statements, do not permit functions as operations on their objects. The above example using limited private types would have an unnatural look to the engineer forcing the use of COPY or ASSIGN procedures. However, limited privates do have their place and should be used when practical.

Practicality also gives the engineer the choice when to make something very "object oriented" or function oriented. Examples are the Generic_Elementary_Functions[2] and the Generic_Linear_Functions. The linear functions for example could have provided very encapsulated private vector and matrix types. An engineer would have gone crazy using the package functions set and get to obtain individual values from the elements of the vector or matrix. Anticipate the common usage of the component, or it will be re-written or unused.

Efficiency led to the creation of many similar functions encapsulated into a single package. A good example of this is the gravity model package, "Gravity." This generic package contains many different models of the earth's gravitational acceleration. It is expected that only one of the models would eventually be utilized after instantiation. This provides the user a selection of fidelity verses speed with little overhead. It would have been possible (and I plan to add one in the near future) to write a routine that would have a parameter selection as to the order of the gravity model. This, however, would have caused inefficiencies for the real time user and minimized the reuse.

Maintainability is the most difficult aspect. The code must be written in a familiar form such as the equivalent to equations in the text books. Documentation must be maintained along with the code. Object oriented techniques did help provide an expandable set of packages from which to work. It also made for the most

difficult decisions, when attempting to implement a different interface/method for a complete generic package. For example, it is not necessary to declare a matrix type just to instantiate the linear functions package if all that is needed was to manipulate vectors. Changing the package to a nested implementation where the matrix was a generic package within the vector packaged caused more changes that I had planned.

Unfortunately (or for those in the business fortunately), maintenance is a never ending task, and attempting to improve the case study by changing the interface/implementation to a different scheme is difficult. There is more than one correct way to implement this code, and perhaps over time the optimal will be discovered.

Validated Compilers

It is interesting to observe the types of bugs that exist in some validated compilers. I am convinced that the validation suite was developed by Atatran coders. Most all of the bugs in the compiler that I have discovered were from the generic implementation of "objects". These errors were not exercised in the validation suite. Some of the choices made during the implementation of this case study were due to work-arounds in the compiler, not due to good engineering practices.

Examples of compiler bugs are worthy of review. I was able to instantiate a generic package to make a function visible. That function would work normally in the routine in which it was instantiated, but I was unable to pass that function as an generic actual parameter to another generic package although functions not declared in instantiated packages did pass as parameters correctly. This initially prevented me from instantiating a linear function package then passing these functions as parameters to other generic functions. Another example was the inability to utilize enumeration types passed as generic actual parameters. I could not even instantiate Text_IO.Enumeration_IO using enumeration types due to these generic problems. In some cases these enumeration problems passed through the compiler and prevented linking or just created code errors. Another example was the inability to pass an array type which was declared in a generic package as a formal parameter into another generic package.

I felt that I was the only person using enumeration types and generics. However, I was extremely pleased by the compiler vendor fixing the problems and sending me a beta copy to continue work. This is the type of support that would be expected on a big budget industrial project, but very happy to say it was also provided to an individual trying to complete a homework assignment.

Other difficulties are cost and capability of compilers on personal computers. This code is not that complex, however, most of the "educational" compilers cannot compile this code due to "out of memory" type errors. More money had to be spent to purchase an upgraded compiler to be able to even compile this code. This code will compile and work well with the R&R Software 386-to-DOS Janus Professional Compiler.

The style of code written in the Ada classroom environment is basically syntax oriented. Code written on a project (even if it is just homework) in an object oriented fashion is much different. Deciding the best methodology for gluing together generic packages requires the engineering work, and not much guidance exists in the literature. At least for the near future, good engineers are required in the code generation process.

The Code

Over 60 pages of code resulted from this effort. Each package was developed to be stand-alone and reusable. I maintain a three ring binder for the documentation each divider containing one package with the requirements, design, code, notes, copies of pages from text books and other references, test programs and results. The packages that tie the code together are the templates which use the package Astro_States as the primary package. Once properly instantiated, the engineer can write application code similar to a MathCad or MatLab program utilizing functions specific to astrodynamics. The advantage is the capability to compile this code so that it can run real time or faster.

This code is centered around a package called Astro_States. This package defines three objects, which contain the state information (position and velocity) in three different frames of reference: Classical Keplerian Elements, Earth Centered Inertial (fixed frame) and Flight Path Coordinates (rotating frame). Figure 2 is two code fragments from Astro_States including the private part describing the states.

These private types have member functions to initialize the states, convert between the states and also get data from the states. Two other functions are time of flight between two states, and a ballistic propagation function given a time. The package also develops the derivatives of the states for use in an integrator.

```
⋮
Package ASTRO_STATES is

 type Index is ( X , Y , Z ,                    – ECI Axes
          Roll , Pitch , Yaw ,                  – Body Axes
          Vel , Azm , FPA , Long , Lat , Rad , Mass , – FPC
          a , e , I , RAAN , Arg_Per , Tr_Anom ,    – CKE
          '1' , '2' , '3' , '4' , '5' , '6' , '7' ) ; – Generic
⋮


private

 subtype BCI_Index is Index range X .. Z ;
 subtype CKE_Index is Index range a .. Tr_Anom ;
 subtype FPC_Index is Index range Vel .. Mass ;
 subtype Body_Index is Index range Roll .. Yaw ;

 subtype Position_Type is Vector ( BCI_Index ) ;
 subtype Velocity_Type is Vector ( BCI_Index ) ;
 subtype CKE_Data_Type is Vector ( CKE_Index ) ;
 subtype FPC_Data_Type is Vector ( FPC_Index ) ;
 subtype Attitude_Type is Vector ( Body_Index ) ;
 subtype Rate_Type     is Vector ( Body_Index ) ;

 type BCI_3D is record
   Position : Position_Type ;
   Velocity : Velocity_Type ;
   Time     : Real ;
 end record ;

 type CKE_3D is record
   CKE_Data_Set : CKE_Data_Type ;
   Time         : Real ;
 end record ;

 type FPC_3D is record
   FPC_Data_Set : FPC_Data_Type ;
   GHA          : Real ;
   GMT          : Real ;
 end record ;

⋮

end states;
end ASTRO_STATES ;
```

Figure 2, Code Fragment from Astro_States

The integrator package currently contains three integrators, an Euler, RK4, and a 6th order predictor/corrector integrator, but will in the future contain others. The derivative function is passed as a generic parameter to the integrator which in turn is used as a generic function back in the Astro_States.propagate package to permit integration of the state over time. The object being integrated is private, allowing the integrators to be used on many different types. Additional generic parameters include specific operations on the type being integrated so that the functions can implement the processing. Additional features include an optional adaptable step size on the RK4 integrator to minimize processing errors. Figure 3 is an example of one fragment of the package specification.

```
⋮
package Integrators is

 Integration_Error : exception ;

⋮


generic

 type Variables is private ;
 type Real is digits <> ;
 with function Deriv ( X  : in Variables;
                       Y : in Real ) return Variables is <> ;
 with function "+" ( X , Y : in Variables ) return Variables is
    <> ;
 with function "*" ( X : in Real ; Y : in Variables ) return
    Variables is <>;

package Euler is
 procedure Integrate_Euler
  (Y_Start  : in    Variables;
   X_Start  : in    Real;
   X_End    : in out Real;
   Step_Size : in out Real;
   Y_End    :   out Variables);

end Euler;
⋮
```

Figure 3, Code Fragment from Integrators

Functions that need to be passed as generic parameters to the Astro_States.Derivative package are Lift, Drag, Thrust, and Gravity. The Lift, Drag, and Thrust functions have typically been coded in one package called Vehicle.

The Gravity package provides for many gravity models including a J2 and J4 model. These functions, given a position vector, provide an acceleration vector describing the magnitude and direction of "down". Since the earth is not perfectly spherical nor is the density

uniform, "down" changes from here to there. This results in changes in the orbits of satellites. This package allows the user to determine the accuracy, or order, of the model. The higher the order of the model, the more processing time is required. In addition to the existing models work is currently underway to develop a tesseral and sectorial model (coefficients are available to about a 50 by 50 model) as a task to run in parallel with the simulation. The high frequency integrator would use a J2 or J4 calculation plus a delta being the difference between the JX model and the 50x50 model until the task completes, at which time the delta is updated.

A linear function package was written as a generic with unconstrained array sizes and enumeration types for the index. This permits access to State_Data.Position(X) in the ECI frame for readability. This makes the linear function package somewhat more complex since the matrices and vectors don't have simple ranges with which to work. Also the package Generic_Linear_Functions has a nested arrangement so that the inner package is: Generic_Linear_Functions.Generic_Vector_Functions. Generic_Matrix_Functions. Initially it was coded such that a matrix type had to exist in order to use just vector functions. The nesting proved to be an adequate simple solution so that matrices did not have to exist for vector only processing. Other evolution changes were the philosophy to treat the linear functions similar to transcendental functions as intrinsic, or force the user to a very strict object oriented philosophy, requiring the passing of many functions as parameters to each package. The feature of overloading tended to influence me to toward an intrinsic implementation, but maintenance and flexibility won out for an object based package. The functions are passed as parameters in generics rather than the package being "withed" in the implementation. Plans exist to continue the nested chain and add Generic_Quaternion_Functions within the .Generic_Matrix_Functions package. A code fragment is shown in figure 4.

```
with LINEAR_FUNCTIONS_EXCEPTIONS;

package Generic_Linear_Functions is
⋮
-- Package Vectors provides functions for the object
"Vector"

generic

 type Index is ( <> );
 type Real is digits <> ;
 type Vector is array ( Index range <> ) of Real ;
 with function sqrt ( X : in Real ) return Real is <>;
 with function arccos ( X : in Real ) return Real is <>;

package GENERIC_VECTOR_FUNCTIONS is

function "+" (Ve  : in Vector) return Vector;
function "-" (Vec  : in Vector) return Vector;
function "+" ( Vec_1 , Vec_2 : in Vector ) return Vector ;
function "-" ( Vec_1 , Vec_2 : in Vector ) return Vector ;
function "*" ( Scalar : in Real ; Vec_1 : in Vector ) return
    Vector ;
function "*" ( Vec_1 : in Vector ; Scalar : in Real ) return
    Vector ;
function "/" ( Vec_1 : in Vector ; Scalar : in Real ) return
    Vector ;
function "*" ( Vec_1 , Vec_2 : in Vector ) return Real ;
function Dot ( Vec_1 , Vec_2 : in Vector ) return Real
    renames "*" ;
function "**" ( Vec_1 , Vec_2 : in Vector ) return Vector ;
function Cross ( Vec_1 , Vec_2 : in Vector ) return Vector
    renames "**" ;
function Angle ( Vec_1 , Vec_2 : in Vector ) return Real ;
function Norm ( Vec_1 : in Vector ) return Real ;
function Unit ( Vec_1 : in Vector ) return Vector ;
procedure Put ( Vec_1 : in Vector ) ;

generic

 type Matrix is array ( Index range <> , Index range <> ) of
    Real;

package GENERIC_MATRIX_FUNCTIONS is

function "+" (Mat_1 : in Matrix) return Matrix;
function "-" (Mat_1 : in Matrix) return Matrix;
function "+" ( Mat_1 , Mat_2 : in Matrix ) return Matrix ;
function "-" ( Mat_1 , Mat_2 : in Matrix ) return Matrix ;
function "*" ( Scalar : in Real ; Mat_1 : in Matrix ) return
    Matrix ;
function "*" ( Mat_1 : in Matrix ; Scalar : in Real ) return
    Matrix ;
function "/" ( Mat_1 : in Matrix ; Scalar : in Real ) return
    Matrix ;
function "*" ( Mat_1 : in Matrix ;
          Vec_1 : in Vector ) return Vector ;
function "*" ( Mat_1 : in Matrix ;
          Mat_2 : in Matrix ) return Matrix ;
function Invert ( Mat_1 : in Matrix ) return Matrix ;
function Transpose ( Mat_1 : in Matrix ) return Matrix ;
procedure Put ( Mat_1 : in Matrix ) ;
⋮
```

Figure 4, Code fragment from Generic_Linear_Functions Specification

To work one of the boundary condition problems for reentry, a quartic root function was required. This resulted in a package called Real_Roots that has closed form solutions to the Quadratic, Cubic and Quartic functions.

The vehicle model required atmospheric data to calculate drag, lift and thrust. A package Atmosphere was developed to provide functions for density, altitude, radius of the earth at given latitudes, mach number and other atmospheric data.

The atmosphere model required table lookup, so a package Tables was developed to do linear interpolation on 1D and 2D tables. Possible future additions to this would be more exotic interpolation techniques and a binary search, verses the linear search currently implemented.

The last two miscellaneous packages are Astro_Constants and Transformations.

Templates were developed for the final applications, see figure 1. The templates instantiate the generics in the proper order, allowing the user code to be written with all of the available objects and number functions. One template instantiates the high fidelity models and the other instantiates real time models. This permits the final application to be up and running quickly. Final adjustments can then be made in the model selection for the application.

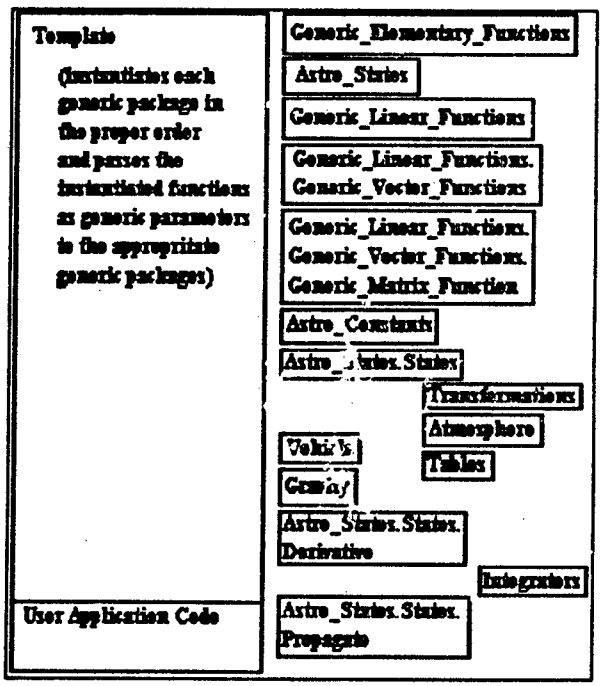


Figure 5, Template

This code is available to students and educators by sending a 3.5 or 5.25 floppy and a SASE to:

Roger Kovacs
2003 S. Evanston Ct.
Aurora, CO 80014

The code is provided for non-profit use, and feedback is greatly encouraged.

Lessons Learned

Determining the object is the most difficult part. Defining the object once identified is simple. This must be an iterative process and should not be constrained by automated tools and processes. A simple change to the main object may and in many cases cause one to discard most all of the previous work. This freedom must be preserved during the first part of a life cycle.

The structure of the program dictates the interfaces and member functions required. This structure has to conform to the limitations of the Ada syntax and compiler bugs. Ada 9X will provide more freedom. This too is an iterative and a learning process. Decisions made include package structure, (object and functional).

Do-all functions may be reusable but in the end may never be reused. Careful attention must be paid to all aspects of reuse. Engineering is an art and science which develops a solution to a problem. The engineer, given guidelines, must have the freedom to select the best solution. Templates provide the user a better starting point for the reuse of the existing code then just comments. It may not be too apparent to a future how to glue together the appropriate packages. Modifications after it is functional are much easier than figuring out how to start from scratch. Reuse is the key, a template is good reuse when the general application doesn't change much.

Embedded test functions or separate test packages provide a simple way to revalidate changes. Most compilers have appropriate switches to eliminate code that does not link in. This then causes no additional overhead to the end user and can provide vital information on the precision and correctness of the package as executed on various platforms.

Conclusion

Ada is an excellent language for the development of reusable packages. The extra overhead required to use Ada over other languages was not noticeable. I was able to keep up or ahead of my colleagues in all of my classes using more of the traditional languages doing the exact same assignments.

The maturity of the compilers is improving, but many of the other languages have an edge. The rejection of Ada as the language of choice for a particular project is more of an attitude and training issue. Quality of Ada code is an experience issue and coding techniques will improve over time. Many of the individuals coding today in industry are engineers of disciplines other than software or computer science. It may be simple to train them to learn Ada syntax, but the larger issue of writing reusable, robust, easily maintained, testable, efficient code will be difficult. This takes Ada (and software engineering life cycle) experience which takes time. I hope to develop over time a balanced blend of two engineering disciplines, Aerospace and Software, and to be able to apply software discipline to aerospace engineering.

It was difficult to find literature in libraries to aid the development of good generic packages, when and where to use private and limited private types and examples or case studies somewhat larger than a few lines of code. Once a standard methodology is developed for aiding in these decisions, Ada will stand out alone in the reusable software market.

This assignment provided me with lessons learned that is documented in this case study. Engineering is a decision making process and many of these decisions were difficult. More experience will aid in the maturity of the code generated. I recognize that this experience only touches a very small part of a software life cycle. It will take more time to grow and learn through experience the many techniques and methods that work and don't work. I will have to see which modules are really reused, and which ones are continually rewritten, and for what reasons.

Future

I envision that in the future, software can be developed with a graphics interface similar to current design tools which will use these and other reusable components, and allow the user to graphically connect the generic parameters. When the user has assembled the reusable components, target code will be generated, and Ada will be produced for documentation or transfer to a different target computer. Software engineers will only be required for developing new reusable components (as well as new and better tools). Just as robotics replaced many blue collar workers, I see computers replacing many white collar workers. I desire to be part of the developing future, not standing back watching it go by.

Challenge

Do it better, and let me know how! I will provide copies of the code into the public domain for non-profit educational use. After graduation, configuration control and maintenance will be provided for some time. (Unfortunately, it would be an overwhelming task to provide notices of error reports to all the users, so just write often.) The evolution of this code should provide a much better case study than the initial release. Let's see what we can all do in a year.

---

[1] Shlaer, S and Mellor, S., OBJECT LIFECYCLES, Modeling the World in States, Yourdon Press, 1992

[2] Generic_Elementary_Functions as proposed by the SIGAda NumWG in 1990.

Biography:
Roger Kovacs is a software test engineer at Martin Marietta Aerospace in Denver, Colorado and is currently completing a ME, Space Operations at the University of Colorado at Colorado Springs. He has a BS in Electrical Engineering Technology from Metropolitan State College of Denver. The only formal undergraduate software education is a single class in PDP-11 assembly language. Self taught languages include Pascal, C, C++, and various assembly languages. Ada was pursued with vigor in a graduate level class at the University of Denver.

Architectural Decomposition of Software Applications

Kimberly Reese and Gary Cort
Los Alamos National Laboratory
Los Alamos, New Mexico

Abstract

Modern software management and quality assurance necessitates development of unique solutions to increasing complexities. One such solution is the concept of representing software applications in hierarchal structures such as file list. The benefits of the file list are listed in detail.

Introduction

Los Alamos National Laboratory Technical Software Management Group employs innovative approaches to develop, maintain, and operate software. The Technical Software Management Group's mission is to efficiently manage all Los Alamos National Laboratory Yucca Mountain Project software applications.

With the increasing complexity of software applications, software components are becoming significantly more difficult to manage. Modern Technical Software Management and software quality assurance approaches require detailed information regarding the organization and interrelationships of the components of complex software applications. Such information is invaluable for defining and cataloging software baselines; performing audits on software development projects; creating packets of information for review; automating the submission, release, and certification of software products; gathering metrics throughout the software project; and assessing the scope and impact of proposed changes to an application.

In order to accomplish these goals, a mechanism or structure representing the architecture of a complex software application must be devised and the development of software tools for processing the architectural information must be implemented. This structure is known to the Technical Software Management Group as a *file list.* When the *file list* is defined and the tools required to process it are in place, the manageability of software applications will increase and the foundation for the development of other software configuration management and software quality assurance tools will be established.

Software Applications

A software application is a collection of files associated with either a reuse component or functionally related computer

programs. These files are divided into primary components and support components. The primary components associated with a particular software application are composed of source code modules and documentation directly related to the application. For example, primary components might consist of functionally related computer programs, data sets required for input to obtain the data sets required for output, and possibly documentation on how to use the program(s). Support components of a software application are a group of modules incorporated into a testing product. The testing product constitutes source code modules and documentation that are indirectly related to the application. Examples of indirectly related modules are testing files, each related to a different test of the source code modules, and documentation, explaining what is being tested. Source code modules are either computer programs or reuse components. The difference between computer programs and reuse components is that functionally related computer programs can perform different tasks alone whereas reuse components are procedures or functions that cannot stand alone and must be incorporated into a program to perform a variety of tasks. Computer programs and reuse components are never combined in the same software application. The software application is comprised of modules directly or indirectly related to the computer programs or reuse components. Documentation shall incorporate significant events of the development of a software application and be documented in text files. These text files, or modules, could be user guides, reports, installation scripts, data set descriptions, modules and methods summaries, software design documents, software requirements specifications or version description documents.

With such a wide variety of modules for a single software application, to manipulate, manage, and keep track of the software application and its modules mentally or physically could cause many errors. Operating on the software application as a single entity is a necessary requirement. It is obvious that an organizational methodology is needed to maneuver software applications. The idea of the *file list* was designed to fulfill this by Gary Cort and Steve Donahue in addition to aid the Technical Software Management Group in managing software applications.

The File List Approach

The purpose of a file list is to identify related components within a software application. A file list is a group of text files. Each group is again a file list, which identifies collections of functionally related modules within a software application. The file list allows modules to be logically associated and controlled as a unit. The file list provides an organized scheme for all software applications. With an appropriate organization of the software application, a hierarchical structure is generated.

An entire software application is represented by a file list(Figure 1). A file list is an abstract notion which decomposes a software application into collections of related files. Each collection of related files is itself a file list. An individual file list(Figure 2) is

a file containing comments and names of interdependent files. Any one of these interdependent file names may also be a name of a file list, representing a cluster of names of interdependent files.

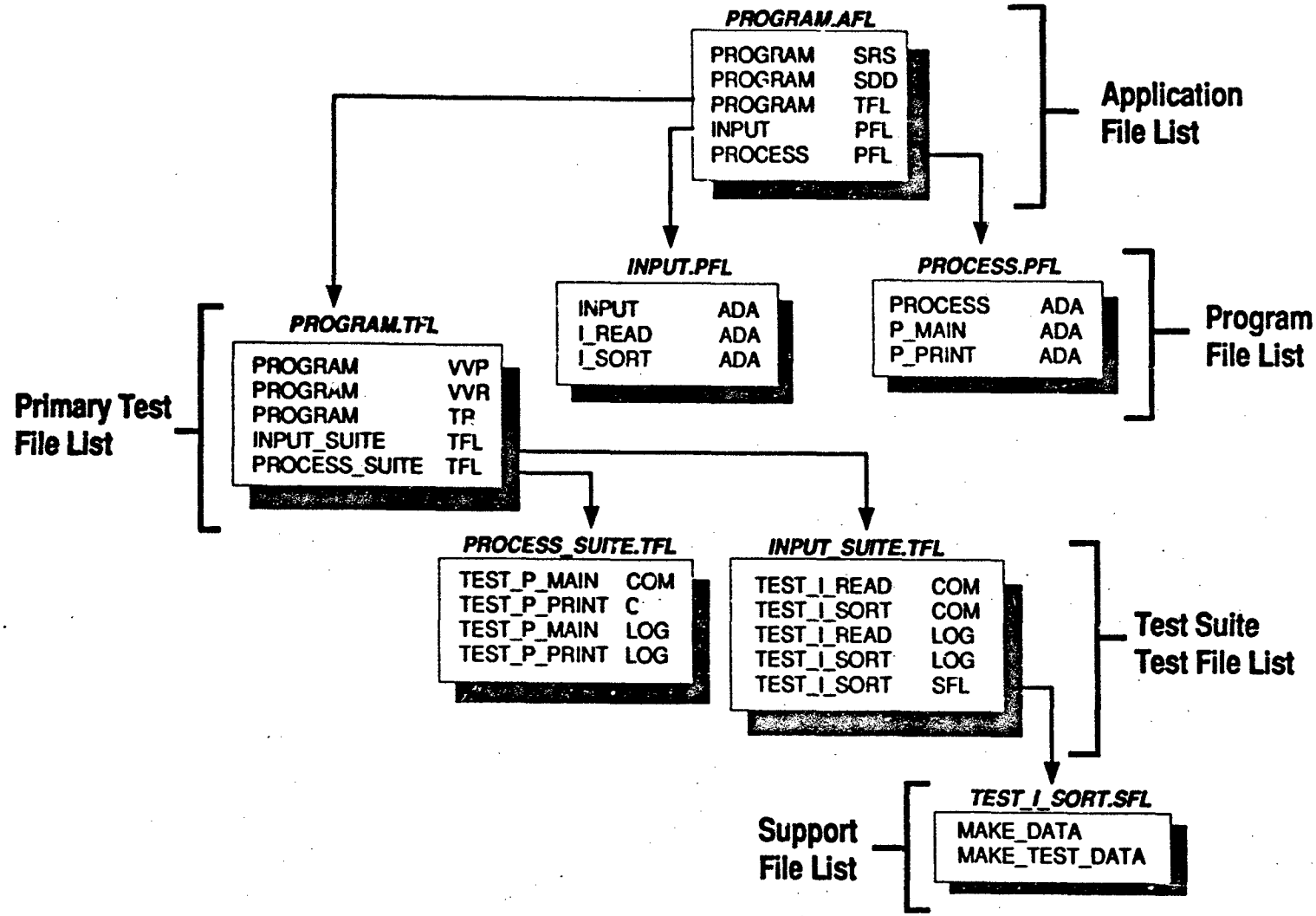
Each text file, or file list, within the file list hierarchical structure specifies a component of the software application. For example, suppose there exists a software application, called "*Program.*" The "*Program*" represents the modules associated with and including the computer programs; *input* and *process*. To convert the software application into a file list, *Program* now becomes *PROGRAM.AFL*. An Application File List (AFL) provides the highest level view of a software application hierarchy. It provides references to modules that are global to the entire application. The *PROGRAM.AFL* is a file consisting of the names of the programs; *INPUT.PFL* and *PROCESS.PFL*, the testing product; *PROGRAM.TFL*, software requirements specification; *PROGRAM.SRS*, and software design documentation; *PROGRAM.SDD*. The architecture of an AFL automatically specifies the dependency relationships between the primary and support components.

The primary components represent the Program File Lists (PFLs) or the Reuse File Lists (RFLs). In figure 1, the Program file list's primary components are the computer programs *INPUT.PFL* and *PROCESS.PFL*. Because the *PROGRAM.AFL* contains computer programs, it cannot contain reuse file lists(RFLs). PFLs and RFLs are separate software applications. PFLs specify the components of the primary software products provided to the users of an application.

There is a one-to-one correspondence between PFLs and user programs. Notice in figure 1 that *INPUT.PFL* and *PROCESS.PFL* have similar names listed within; *INPUT.ADA* and *PROCESS.ADA*. These are the names of the user programs. Other program names listed within, i.e. *I_READ.ADA*, are support programs. RFLs specify the components of the primary reuse products provided to the users of an application. There is a one-to-one correspondence between RFLs and user packages.

The support components in the hierarchical structure are the primary Test File Lists(TFLs). The TFLs are major components of the overall testing effort and represent the testing product. The primary Test File List shall reference zero or more Test Suite Test File Lists(TFLs) and Support File Lists(SFLs). The primary TFL will also reference the verification and validation report; *PROGRAM.VVR*, verification and validation plans and procedures; *PROGRAM.VVP*, and test results; *PROGRAM.TR*. Test Suite Test File Lists and Support File Lists are organized hierarchically beneath the primary TFL to reflect the internal structure of the testing effort. Test Suite TFLs should be listed in the primary TFL. The two test suite TFLs listed in figure 1 are *INPUT_SUITE.TFL* and *PROCESS_SUITE.TFL*. Suite is another word for a collection of related files. Itemized within are computer programs and subprograms that are used for testing. These are only two testing aspects of the software application. There can be many Testing Suite TFLs for one program in order to verify that the program works correctly and will not crash. There is no limit to the number of Test Suite Test File Lists

Figure 1

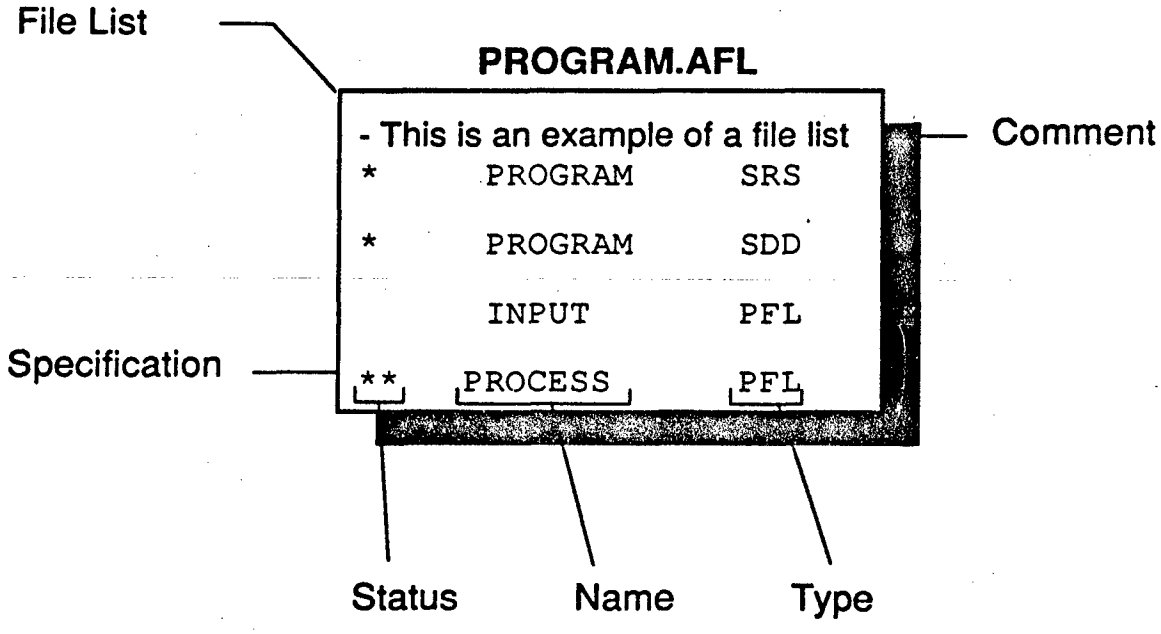


This hierarchical organization of a *file list* that illustrates the decomposition of a program software application.

or Support File Lists that may be specified for an application. A TFL at any level may reference Support File Lists. The one Support File List shown is *TEST_I_SORT.SFL*. This contains information needed in order to run testing in *INPUT_SUITE.TFL*. Support File Lists (SFLs) are summarized auxiliary programs that provide additional resources (test data, special environments, etc.) for the testing effort. There is a one-to-one correspondence of SFLs to support programs. All file lists have a specific format that also aids configuration management's efforts to regulate software.

File list structures are made up of entry lines of information, either comments or specifications (Figure 2). The file list structure has a unique name and a type. Illustrated in figure 2, the unique name is *PROGRAM* and type is *AFL*. A comment, delimited by a hyphen in the left margin, contains explanatory information. This information could be the name of the software application, the description of the software application, or anything related to the software application. The comment is not processed by any file list operations. It is only needed for user communication. A specification characterizes a single module in terms of the component's name, type, and status. The name is the primary identification attribute of a specification. It corresponds exactly to the file name of the associated application component on the host file system. The type is the extension of the associated application component. It corresponds to the type of file it is. In

Figure 2



This *program application file list* is an example of how a *file list* is decomposed.

figure 2 examples of names are *PROGRAM, INPUT* and *PROCESS* and types are *SRS, SDD*, and *PFL*. The status specifies the change processing status attribute of an application component through the presence (or absence) of a standard prefix. This attribute may assume one of three possible values: NEW, MODIFIED, or OLD. The status of a component is MODIFIED (prefix = *), if a currently approved version is undergoing change. NEW status (prefix = **) denotes a module for which no previous approved version exists. Stable existing modules are assigned the OLD status (no prefix).

Processing a File List

Now that the design has been laid, the tools to process the file list must be implemented. These tools were incorporated into a software package. The tools included: *VISIT_ALL, VISIT_MODIFIED, VISIT_NEW, VISIT_OLD, PARENT_OF, ANCESTRY_OF, NAME_OF, STATUS_OF*, and *TYPE_OF*. These tools allowed Technical Software Management Group to efficiently control software applications with minimum overhead.

Each tool has a different function. The *VISIT_ALL* gave the ability to iterate through an entire software application(file list). For instance, *VISIT_ALL* would start at the top level, *PROGRAM.AFL*, and return one entry at a time, *PROGRAM.SRS, PROGRAM.SDD*, and *PROGRAM.TFL* (from figure 1). When *PROGRAM.TFL* is reached , the program then branches off to iterate through *PROGRAM.TFL*, returning PROGRAM.VVP, PROGRAM.VVR and then *INPUT_SUITE.TFL*. When it reaches *INPUT_SUITE.TFL*, it repeats the process again until the entire file list is iterated. The *VISIT_MODIFIED* procedure gives the ability to iterate through only the file names that have been modified. which means that it returns only modified entries. This is the same for *VISIT_NEW*, and *VISIT_OLD*, which return new and old entries respectively. The *PARENT_OF* function returns the name of the file that the file resides in. For example, the parent of *PROGRAM.VVP*, in figure 1, is *POGRAM.TFL*, and the parent of *P_PRINT.ADA* is *PROCESS.PFL*. The *ANCESTRY_OF* function returns the entire list of ancestors not including the file that the ancestry was to be found of. For instance, the ancestry of *MAKE_DATA* would be *PROGRAM.AFL/PROGRAM.TFL/INPUT_SUITE.TFL/TEST_I_SORT.SFL*. The functions *NAME_OF, STATUS_OF*, and *TYPE_OF*, would return the name status and type respectively. Looking at figure 2, the name of *PROCESS.PFL* would be *PROCESS*, the status would be NEW, and the type would be PFL.

The automation of this information eliminates routine, lengthy, and inaccurate operations. These tools supply the data necessary for the development of other software configuration management and software quality assurance tools.

The Benefits of a File List

The file list provides a precise and clear understanding of the organization of a software application. The tools used to process a file list can be employed to obtain useful information. For example, a printout of a file list can easily supply the directory

information to store a software baseline, or a collection of software components. When a very large software project is to be developed, a file list complements the requirements and design document of the project and clarifies exactly what is to be accomplished. A file list can be held as a check list for the physical and functional requirements of software. For example, if the developer turns in a baseline, the configuration manager can cross check each software component handed in with the file list. This method assists in tracking complete and needed to be completed software and documentation. The file list allows assembling and printing of the review packets of information, which could contain over a 1,000 different files, to be automated. This process can be easily automated with the VISIT_ALL procedure, which then would requite only the file list name to locate, validate and transfer all of the files associated with the baseline; therefore, eliminating the tedious effort to type each file name, and quickly creating a more accurate packet of information.

A file list structure is also helpful for software updates. It would not be beneficial to release an entire software application with over a 1,000 software components to a developer for updating, when only a few files need to be modified. A file list allows for files to be marked effectively, using the status; NEW, MODIFIED, and OLD. The tools can be used to ensure that only the necessary files are sent to the developer. Since the developer only has access to the software components related to the update, the configuration manger is still in control of the software application. Maintaining control of the software application, is important for greater manageability.

File list structures also aid in gathering metrics throughout the software project life cycle. Metrics aid software configuration management in defining good, reliable software. The process of the software is also important. For instance, the number of lines of code in a software component can be used to measure the efficiency of the process. The tedious task, to manually count each line of code would take many hours and include many human errors. This task can be automated by the use of file list. By automating the process, results will be quick, efficient, and accurate.

Conclusion

The complexity and magnitude of today's software projects necessitates an enhanced, organized, and logical structure, *file list.* File lists are easy to maintain and manage. Using file lists enhances the processes of classifying and characterizing software baselines, verifying the requirements of software development projects, producing review packets, automating the modification of software products, obtaining measurements, and determining the range of impact of proposed changes. The file list insures a consistent hierarchical organization for the decomposition of a complex software application. The automation of these tools will assure reliable, precise, and rapid results that will relieve the poor soul of the burden to be a slave to the key board. The Technical Software Management Group at LANL has put the file list in place and created some of the tools mentioned. They now work on other tools that may assist them in reducing tedious

and boring operations and improving efficiency.

References

1. Cort, Gary and R. O. Nelson. "The Los Alamos Software Development Tools." Proceedings of the Digital Equipment Computer Users Society (1985): 391-94.

2. Cort, Gary and R. O. Nelson. " The Los Alamos Tool-Oriented Software Development System." Proceedings of the Digital Equipment Computer Users Society (1985): 395-99.

3. Donahue, Steve and Gary Cort. "The LANSCE Software Management Environment." Proceedings of the Digital Equipment Computer Users Society (1988): 127-39.

4. Software Configuration Management. LANL-YMP-QAPP Los Alamos National Laboratory Quality Assurance Plan for the Yucca Mountain Project.

5. Software Configuration Management. File List Standards.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy, Los Alamos National Laboratory and with the aid of Gary Cort and Donn Hines.

About the Author:

Kimberly E. Reese
Stockton State College
Pomona, NJ 08240
stk1054@vax003.stockton.edu

Kimberly E. Reese is presently a student at Stockton State College, working on a B.S. in both mathematics and computer science. Her primary objective is to obtain a Ph.D. and do research and development in the computer science field. She has been successful working as a cooperative education student for the Federal Aviation Administration. Her work for the Operation Research and Analysis Branch entailed studies of aviation system safety and efficiency. More recently she worked as an intern for Los Alamos National Laboratory, Earth and Environmental Sciences Division. As a software engineer, she developed a reuse package in ADA to assist in the management of software. Presently, she works with the Federal Aviation Administration, in Human Factors Division, developing an user interface.

Teaching the Second Course in Computer Science in a Reuse-Based Setting: A Sequence of Laboratory Assignments in Ada*

Jeff Gray
jgg@cs.wvu.wvnet.edu

Department of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506

Correspondence: Murali Sitaraman - murali@cs.wvu.wvnet.edu, (304)-293-3607

* This research is funded in part by DARPA Grant # DAAL03-92-G-0412 awarded to West Virginia University and Muskingum College, New Concord, Ohio.

Abstract

We are currently exploring a new approach for introducing key software engineering and computer science principles in the second course of our curriculum. Our approach introduces software reuse as a context for providing a motivation toward learning the importance of principles such as abstraction, specification and design. An essential element for the success of the reuse-based approach is an appropriate series of lab assignments. First, we present the students with components designed and implemented by the lab instructor. Based only on the specifications, students learn to assemble these components and solve interesting problems. Later, students reuse these components to build layered implementations of other components. Only toward the end of the course are they taught how to write their own implementations from scratch, such as using pointers. This paper describes three lab assignments which illustrate our approach.

1 Introduction

Why teach reuse in an introductory course?

Most current undergraduate computer science curricula suffer from two fundamental problems, which often lead to several others. One problem is the absence of a context, and hence motivation, for learning fundamental principles of computer science (e.g. abstraction) in the second course. The other problem is late exposure to principles of software engineering, such as those found in a senior level course, resulting in relatively inexperienced graduates in applying these principles.

We believe these problems can be ameliorated by providing software reuse as a context in which to teach the fundamental principles of computer science. The reuse-based approach also motivates key software engineering principles that are often omitted when the course is taught outside of this context. The principles instilled by the lab assignments discussed in this paper include:

- -- The ability to understand abstract and formal specifications:
- -- Specification-based component reuse;
- -- Separation of the specification of a component from its implementation;
- -- Construction of new components by layering them on top of existing components;
- -- Multiple implementations (with different efficiency characteristics) for a given specification.

Introduction to these principles early in their undergraduate careers will give students ample time to gain confidence in their abilities by applying the principles throughout their remaining courses. This paper describes an approach toward the construction of laboratory assignments which attempt to meet the above goals. During the past year, such assignments have been used in a section of the second semester freshmen level computer science course at the West Virginia University.

Our definition of reuse

Recent literature on software reuse contains several different definitions or classifications of the term [6]. The definition of reuse used in this paper is component-based [8, 11]. We view a reusable component as having two distinct elements: a formal specification and a certifiable implementation of that specification, possibly in the form of object code. All references to reuse discussed in this paper are based only on the specification and performance characteristics (e.g. performance efficiency) of the implementation. We concentrate on components which are *designed* for reuse and are not concerned with definitions of the term that deal with code scavenging or other methods where the utilization of already existing software occurs by accident or serendipity.

Organization of the paper

The paper is contained in five sections. Sections two through four describe aspects of different laboratory assignments that have been used to introduce students to software engineering and reuse principles. Each of these sections contains the goals, descriptions, and possible variations on the theme of a particular lab. The assignments chosen to be discussed in these sections are only a subset of the total collection of labs that we have developed but are those which best exemplify our overall goals. A final section summarizes the paper and offers suggestions for possible areas of future work.

2 Student as client of a reusable component

Goals

To teach the following principles:

-- The ability to understand formal and abstract expressions of a specification;

-- Specification-based component reuse;

-- The need for separation of the specification of a component from its implementation;

-- Acquaint the students with the notation of a specification language;

-- Construction of secondary operations.

Description

The purpose of this laboratory assignment is to solve a backtracking problem iteratively using a stack package provided by the instructor. Several different backtracking problems have been introduced to the students. Examples of problems we have used in the past include:

-- The Eight Queens problem, whereby the students must find all possible combinations of placing eight queens on a chess board so that no queen can be attacked by another;

-- Helping a mouse find a piece of cheese by moving through a maze which contains dead-ends;

-- Assisting a squirrel in climbing to the top of a tree, filled with many empty branches, to find an acorn.

All of the above problems share a common trait in that a decision must be made to explore down one of several paths. Also, an ability must be provided so that one can backtrack to previous spots and choose alternative paths when dead-ends are encountered.

The students are asked to solve these problems iteratively using a stack. The specification of an Ada package that provides a stack component is shown in Figure 1. The students are given a copy of this specification and told how to access the object code version of the body to allow for proper linking. They must construct a client program which utilizes the stack package to solve the backtracking problem. The client program is then linked with the stack package to obtain an executable.

When the students are given the stack component, they are asked to view the specification as a contract between themselves and the implementer of the package (i.e. the lab instructor). This reinforces the notion that the developer and user of a component are often different people. They are assured that the stack operations will work correctly provided they follow the specification. They must surmise on their own, by reading the specification, the syntax and meaning of each operation. Thus, the students get an early example of the importance of providing specifications which are unambiguous. To add semantic information to Ada package specifications, we use a close dialect of the RESOLVE specification language [4, 10,

11]. RESOLVE specifications are formal, but yet succinct and understandable by freshmen who have been briefly exposed to topics covered in discrete mathematics.

In Figure 1, the type Stack is modeled as a mathematical string. Manipulations on a variable of type Stack are described using functions borrowed from mathematical string theory (e.g. the concatenation operator = "o", found in the Push and Pop operations). Operations are specified using a requires clause (pre-condition) and an ensures clause (post-condition). Some operations may not have a requires clause. These clauses are mathematical assertions and not executable statements. The requires clause states what needs to be true before the operation is called while the ensures clause states what the operation will do provided the requires clause is satisfied at call time. A call to an operation without satisfying the requires clause is undefined and can do anything. Each reference to a variable in the requires clause refers to the value of the variable at the time the procedure was invoked. In the ensures clause, however, the value of a variable at the time of procedure invocation is accessed by preceding the variable name with a '#' sign. Reference to a variable without the '#' sign refers to its value at the time the operation returns to the caller. Also, since most character sets, including ASCII, do not provided symbols for the universal quantifiers or lambda (i.e. the empty string in our specification), we resort to spelling out the definition of these symbols rather than giving the symbol itself. Aside from an explanation of the concatenation operator, which should be familiar to most readers, the above discussion provides an individual with enough detail to comprehend the meaning of each operation. One can simply view the stack operations as manipulations on a string whereby calls to the Push operation "consume" an item and place it at the end of a string (i.e. S = #S o X); the returned item is assigned an initial value depending on the type of the item. Calls to the Pop operation remove an item from the end of the string (i.e. #S = S o X).

The Basic_Stack_Template has been designed for reuse by following guidelines such as those in [4]. There are several differences between specifications designed using these guidelines and other specifications found in discussions like [1]. Details of these design issues are beyond the scope of this paper. An interested reader is referred to [2, 4, 5, 8, 11] for more detailed descriptions of design issues such as why standard operations called Initialize, Finalize, and Swap are provided for every type.

```
generic

  type T is limited private;

  with procedure T_Initialize(X : in out T);
  --! ensures T.Init(X)

  with procedure T_Finalize(X : in out T);

  with procedure T_Swap(X, Y : in out T);
  --! ensures (X = #Y) and (Y = #X)

package Basic_Stack_Template is

  type Stack is limited private;
  --! type Stack is modeled by a string of T...

-- standard operations...

  procedure Initialize(S : in out Stack);
  --! ensures S = Lambda

  procedure Finalize(S : in out Stack);

  procedure Swap(S1, S2 : in out Stack);
  --! ensures (S1 = #S2) and (S2 = #S1)

-- primary Stack operations...

  procedure Push(S : in out Stack;
                 X : in out T);
  --! ensures (S = #S o X) and T.Init(X)

  procedure Pop(S : in out Stack;
                X : in out T);
  --! requires S /= Lambda
  --! ensures #S = S o X

  function Is_Empty(S : Stack)
                              return Boolean;
  --! ensures Is_Empty iff S = Lambda

private

  type Representation;
  type Stack is access Representation;

end Basic_Stack_Template;
```

Figure 1
Specification of a Stack Component

A final requirement of the assignment is to construct what are termed secondary operations for the component. Secondary operations provide additional functionality in using a particular

component. These operations are often not included in the list of primary operations due to the fact that they can be implemented efficiently without underlying knowledge of the abstract data type representation. To illustrate the difference, the implementation of the primary stack operation called Push must have access to the underlying representation of a stack in order to properly add an element. It needs to know whether the stack is being represented using pointers, arrays, or layered on top of some other component. However, a secondary operation called Copy_Stack, for instance, does not need access to the representation and can be written simply using a loop with proper calls to the primary operations Pop and Push, in addition to the possible need of temporary variables.

The assignment directs the student in assembling two secondary operations for stacks. The operations that the student must write are Reverse_Stack and Print_Stack. These operations are needed in the assignment to print the actual solutions to the problem that the client program discovers. Since the secondary operations require access to various primary operations, the needed primary operations must be passed as generic parameters. An example of how this might be accomplished is found in Figure 2. All the standard (i.e. Initialize, Finalize, and Swap) and primary operations of both the element type T and the Stack type are passed as generic parameters.

```
generic

 -- Semantic specifications for the following
 -- operations are the same as those found in
 -- Figure 1.

 type T is limited private;

 with procedure T_Init(X : in out T);
 with procedure T_Fin(X : in out T);
 with procedure T_Swap(X, Y : in out T);
 with procedure T_Print(X : in out T);

 type Stack is limited private;

 with procedure Initialize(S : in out Stack);
 with procedure Finalize(S : in out Stack);
 with procedure Swap(R, S : in out Stack);
 with procedure Push(S: in out Stack;
                     X: in out T);
 with procedure Pop(S: in out Stack;
                    X: in out T);
 with function  Is_Empty(S: in Stack)
                        return Boolean;

package Secondary_Stack_Ops is

-- secondary operations...

  procedure Reverse_Stack(S : in out Stack);
  --! ensures S = #S^R

  procedure Print_Stack(S : in out Stack);
  --! ensures (S = #S) and (output = S)

end Secondary_Stack_Ops;
```

Figure 2
Specification of a Component for Secondary Stack Operations

Variations

As stated above, there are three variations to the backtracking problem which we have used as laboratory assignments. Similar labs that make use of abstract data types other than a stack could be developed. For example, a lab instructor might give the students a queue package and ask them to write a client program that uses the component. They might be asked to use the queue to simulate a message passing system where requests to send and receive messages are handled and placed on a queue. Alternatively, they might use the queue to simulate a row of tellers at a bank where each teller has a queue of customers with individual requests to be serviced.

3 Student as an implementer of a layered component

Goals

This lab instills the following principles, in addition to those already named in section two:

- Construction of new components by layering them on top of existing components;
- Multiple, plug-compatible, implementations (with different efficiency characteristics) for a given specification.

Description

This section describes an assignment that is along the same idea as the last section but offers somewhat of a change in the implementation of the stack package. In this assignment, the students are given the specification to a list component shown in

Figure 3. Implementation details about this component are hidden but access to the object code is provided to allow linking. They are then asked to use this component to actually implement the operations of the stack package which they have already seen and used. They must implement the stack operations solely by making calls to the operations of Figure 3 and are not allowed to use any form of pointers or array constructs. Thus, a stack package is implemented by layering it on top of another component. The lab described in the previous section is reused in this case by re-linking it with the new stack implementation. The assignment should assist the students in beginning to think about how multiple implementations for the same specification are constructed (see [9]). Also, the ease with which this lab can be completed should reinforce the idea of reuse. Students learn that it is often advantageous to make use of pre-existing standard components rather than "re-inventing the wheel".

The concept used to represent the list component in Figure 3 is different from the typical list concept presented in textbooks like [1]. In particular, the abstract idea of lists is presented without discussing pointers or access types. A type called list is modeled as two strings of some other type T. These two string are called, appropriately, "left" and "right". This view can be better understood if one envisions a conceptual cursor that separates the two strings. The package provides operations to move this cursor around the list as well as the ability to perform insertions and deletions. To illustrate this notion of a cursor, as it would apply to a list, examine the following instance of a list variable called L:

```
      |
3 4 | 7 2 6 3
      |
```

The value of L.Left would contain the two elements 3 and 4 while the value of L.Right would contain the four values 7, 2, 6, and 3. All insertions and deletions are performed to the right of the cursor. The Reset and Advance operations are used to traverse through the list. Using the above values of list L, a call to the Reset operation, followed by a call to Remove would result in L now resembling the following:

```
      |
| 4 7 2 6 3
      |
```

As a design principle, functions needed to check the requires clause of all operations are also included in the specification (i.e. function At_Right_End). The operation Swap_Right will not be used in this assignment. It has been provided for future assignments that may implement secondary operations since it has been found useful in constructing efficient implementations of a Copy_List operation [11].

The students have often found that this assignment can be completed within several hours. Almost all of the required stack operations that they must write can be implemented with merely one line of code. For example, code to implement the Push operation would simply entail making the proper call to a corresponding list operation (i.e. Insert). Similar reasoning follows for the other stack operations provided the students take care to preserve the FIFO ordering of the stack. A student only needs to understand the specification of the list component well enough to discern what calls correspond to similar notions within the stack operations. This reinforces the concept of specification-based component reuse.

```
generic

  type T is limited private;

  with procedure T_Initialize(X : in out T);
  --! ensures T.Init(X)

  with procedure T_Finalize(X : in out T);

  with procedure T_Swap(X, Y : in out T);
  --! ensures (X = #Y) and (Y = #X)

package Lists is

  type List is limited private;
  -- type List is modeled by a pair of strings of T,
  -- named Left and Right

-- standard operations

  procedure Initialize(L: in out List);
  --! ensures (L.Left = Lambda) and
  --          (L.Right = Lambda)

  procedure Finalize(L: in out List);

  procedure Swap(L1, L2: in out List);
  --! ensures (L1 = #L2) and (L2 = #L1)
```

Figure 3
Specification of a List Component

```
-- primary List operations

procedure Reset(L: in out List);
--! ensures (L.Left = Lambda) and
--!         (L.Right = #L.Left o #L.Right)

procedure Advance(L: in out List);
--! requires L.Right /= Lambda
--! ensures :
--! (L.Left o L.Right = #L.Left o #L.Right) and
--!  (thereExists x: T, s.t., L.Left = #L.Left o x)

function At_Right_End(L: in List)
                        return Boolean;
--! ensures At_Right_End iff L.Right = Lambda

procedure Insert(L: in out List;
                 X: in out T);
--! ensures (L.Left = #L.Left) and
--! (L.Right = X o #L.Right) and T.Init(X)

procedure Remove(L: in out List;
                 X: in out Item);
--! requires L.Right /= Lambda
--! ensures (L.Left = #L.Left) and
--! (#L.Right = X o L.Right)

procedure Swap_Right(L1 : in out List;
                     L2 : in out List);
--! ensures (L1.Left = #L1.Left) and
--! (L2.Left = #L2.Left) and
--! (L1.Right = #L2.Right) and
--! (L2.Right = #L1.Right)

private

 type Representation;
 type List is access Representation;

end Lists;
```

Figure 3 (cont.)
Specification of a List Component

Variations

Although the above description layers a stack package on top of a pre-existing list component, it is certainly plausible that one could also use alternative abstract data types. For instance, the students might be asked to implement a stack layered upon a deque or a standard FIFO queue rather than a list. They also could be asked to analyze the efficiency of each operation in comparison to other strategies. As an example using a FIFO queue to build a stack, if the push operation executes in constant time, then the pop operation must run in linear time due to the need to retrieve the element at the end of the queue since the ordering of the two data structures (i.e. FIFO versus LIFO) differs.

4 Student as an implementer of a reusable component built from-scratch

Goals

In addition to the principles named in sections two and three, this lab introduces the following new concept:

-- Use of access types to efficiently implement components from-scratch.

Description

This section describes variations to a laboratory assignment that is often presented toward the end of a semester. It tends to focus more on specific details of implementing components (e.g. using pointers). It builds upon the previous two discussions by requiring the students to finally write lower level implementations of the list component. The stack package will still be layered on top of the list but in this case the students acquire a feel for using access types to represent unbounded components.

Variations

Several possible variations could be suggested toward implementing the list in ways other than pointers. The list itself could be layered upon an already assembled component or the implementation details might opt to focus on an array based approach. Additionally, rather than concentrating on using a list to construct the stack as in section three, the idea of pointers could be used to implement the stack directly which would allow one to eliminate the need for implementing lists altogether. Also, secondary operations for lists could be requested similar to those described in the first assignment. Students might be asked to implement a secondary operation which performs a Copy_List, using the primary Swap_Right operation, from one list variable to another variable. Correspondingly, the students may be asked to write secondary operations for the list package to provide the facilities for printing and reversing lists.

A proviso could be added to the assignment which states that all primary operations need to be written in constant time.

This would be mentioned in conjunction with a statement reminding them that the implementer and client of a component are often different individuals. With this in mind, the students will come to realize the need for efficient implementations since the client will probably decide to rewrite the component themselves from scratch if the component does not meet their performance requirements. In this paper we do not go into any details on how the list operations are constructed in constant time but additional information on implementing unbounded reusable components can be found in [3].

5 Conclusions

The structure of most current curricula tends to introduce the fundamental principles of computer science void of any particular context. An introductory course based on a software reuse setting would assist in providing a needed context to introduce these principles. Early exposure to these principles would aid students in applying the ideas toward a vast majority of the programming projects that they would encounter throughout the remainder of their undergraduate careers.

In this paper we presented one approach toward providing a context for teaching the fundamental principles of computer science. With our approach, laboratory assignments are used to inculcate the fundamental principles of computer science whereby software reuse is used as a primary motivator. As examples, a subset of our laboratory assignments currently used at the West Virginia University were described. These assignments first require the student to become a client of reusable components. Later in the semester they are given the opportunity to actually implement their own components at a lower-level (e.g. using pointers).

There is still much work that needs to be done with the implementation of our approach. For example, most of the proposed laboratory assignments that were mentioned under the *Variations* sections need to be constructed. We are also currently working toward conducting a survey to determine the impact of the reuse-based approach as being applied by previous students in other courses in our curriculum.

Acknowledgments

I am indebted to my advisor, Murali Sitaraman, for the help he has offered in completing this paper. His invaluable suggestions were always beneficial whenever I found myself at a crossroad.

Selected References

1. Booch, G., *Software Components with Ada*, Benjamin/Cummings, Menlo Park, CA, 1987.

2. Edwards, S., *An Approach for Constructing Reusable Software Components in Ada*, IDA Paper P-2378, Institute for Defense Analyses, Alexandria, VA, September 1990.

3. Hollingsworth, J.E. and Weide, B. W., "Engineering 'Unbounded' Reusable Ada Generics," *Proceedings of the Tenth National Conference on Ada Technology*, ANCOST, Inc., Arlington, Virginia, February 1992, pp. 82-97.

4. Hollingsworth, J.E., "Software Componenet Design-for-Reuse: A Language Independent Discipline Applied to Ada", Ph.D. dissertation, Deptartment of Computer and Information Science, The Ohio State University, Columbus, OH, 1992.

5. Harms, D.E. and Weide, B.W., "Copying and Swapping: Influences on the Design of Reusable Software Components", *IEEE Transactions on Software Engineering* 17, 5, May 1991, pp. 424-435.

6. Krueger, C.W., "Software Reuse", *ACM Computing Surveys*, Vol. 24, No. 2, June, 1992, pp. 131-184.

7. Muralidharan, S., and Weide, B. W., "Should Data Abstraction Be Violated to Enhance Software Reuse?," *Proceedings of the Eighth National Conference on Ada Technology*, Atlanta, GA, March 1990, pp. 515-524.

8. Muralidharan, S. and Weide, B.W., "Reusable Software Components = Formal Specifications + Object Code: Some Implications", *Third Annual Workshop: Methods and Tools for Reuse*, Syracuse, NY, June 1990.

9. Sitaraman, M., "A Class of Programming Language Mechanisms to Facilitate Multiple Implementations of a Specification", *Proceedings of the 1992 IEEE International Conference on Computer Languages*, San Francisco, CA, April 1992.

10. Sitaraman, M., Welch, L., and Harms, D.E., "On the Specification of Reusable Software Components", *International Journal of Software Engineering and Knowledge Engineering*, 3, June, 1993, to appear.

11. Weide, B.W., Ogden, W.F, and Zweben, S H, "Reusable Software Components", *Advances in Computers*, M.C. Tovits, ed., Academic Press, Vol. 33 (1991), 1-65.

Biography

Jeff Gray can be reached at 1027 Grandview Rd., Glen Dale, WV, 26038. He holds a Bachelor of Science (1991) degree from West Virginia University and is currently pursuing the Master of Science degree.

DOMAIN SPECIFIC SOFTWARE ARCHITECTURES: A PROCESS FOR ARCHITECTURE-BASED SOFTWARE ENGINEERING

Christine Braun
Raymond Coutant
GTE Federal Systems
15000 Conference Center Dr.
Chantilly, VA 22021

James Armitage
GTE Communications Systems
Resident Affiliate, Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Summary

GTE is the Command and Control domain contractor for DARPA's Domain Specific Software Architectures program. The objective of this program is to develop and demonstrate an architecture-driven, component-based capability for the automated generation of command and control (C2) applications. Such a capability will significantly reduce the cost of C2 application development and will lead to improved system quality and reliability through the use of proven architectures and components. This paper describes GTE's approach to the program, focusing in particular on the domain-specific reuse-based software lifecycle.

The DSSA Concept

DSSA is based on the concept of an accepted generic software architecture for the target domain. As defined by DSSA, a software architecture describes the topology of software components, specifies the component interfaces, and identifies computational models associated with those components. The architecture must apply to a wide range of systems in the chosen domain; thus it must be general and flexible. It must be established with the consensus of practitioners in the domain.

Once an architecture is established, components that conform to the architecture—i.e., that implement elements of its functionality in conformance with its interfaces—will be acquired. They may be acquired by identifying and modifying (if required) existing components or by specifically creating them.

The existence of a domain-specific architecture and conformant component base will dictate a significantly different approach to software application development. The developer will not wait until detailed design or implementation to search for reuse opportunities; instead, he/she will be driven by the architecture throughout. The architecture and component base will help define requirements and allow construction of rapid prototypes. Design will use the architecture as a starting point. Design and development tools will be automated to "walk through" the architecture and assist the developer in the selection of appropriate components. The ultimate goal is to significantly automate the generation of applications. A major DSSA task is to define such a software lifecycle model and to prototype a supporting toolset.

These activities are accompanied by extensive interaction with the development community for the target domain, and by technology transition activities. One aspect of this is that each domain team is working closely with a DoD agency that carries out major developments in the designated area. The GTE team is working with the US Army Communications and Electronics Command.

Why Command and Control?

There are many reasons why the command and

control domain is an excellent target for DSSA technology. It is a high payoff area; command and control systems are needed even in the current military climate. (This is particularly true when one recognizes that applications such as drug interdiction and emergency relief fall within the C2 "umbrella".) It is a well-understood area; most of the processing performed in C2 applications is not algorithmically complex. However, C2 applications are very large, and much of this size comes from repeated similar processing—for example, parsing hundreds of types of messages. In addition to this commonality within applications, there is much commonality across applications. Multiple C2 systems must handle the same message types, display the same kinds of world maps, etc.

The kinds of commonality in C2 applications are very well-suited to DSSA techniques. In some areas, components can be reused identically; these can be placed in the DSSA component base and highly optimized. In other areas, components will be very similar in nature but differ in the particulars, e.g., message parsing. These areas are a natural fit to the DSSA component generation technology, allowing a table-driven generator to quickly create the needed specific component instances.

GTE's Approach

Figure 1 illustrates GTE's overall approach to the DSSA program.

Initially, project work follows two parallel threads.

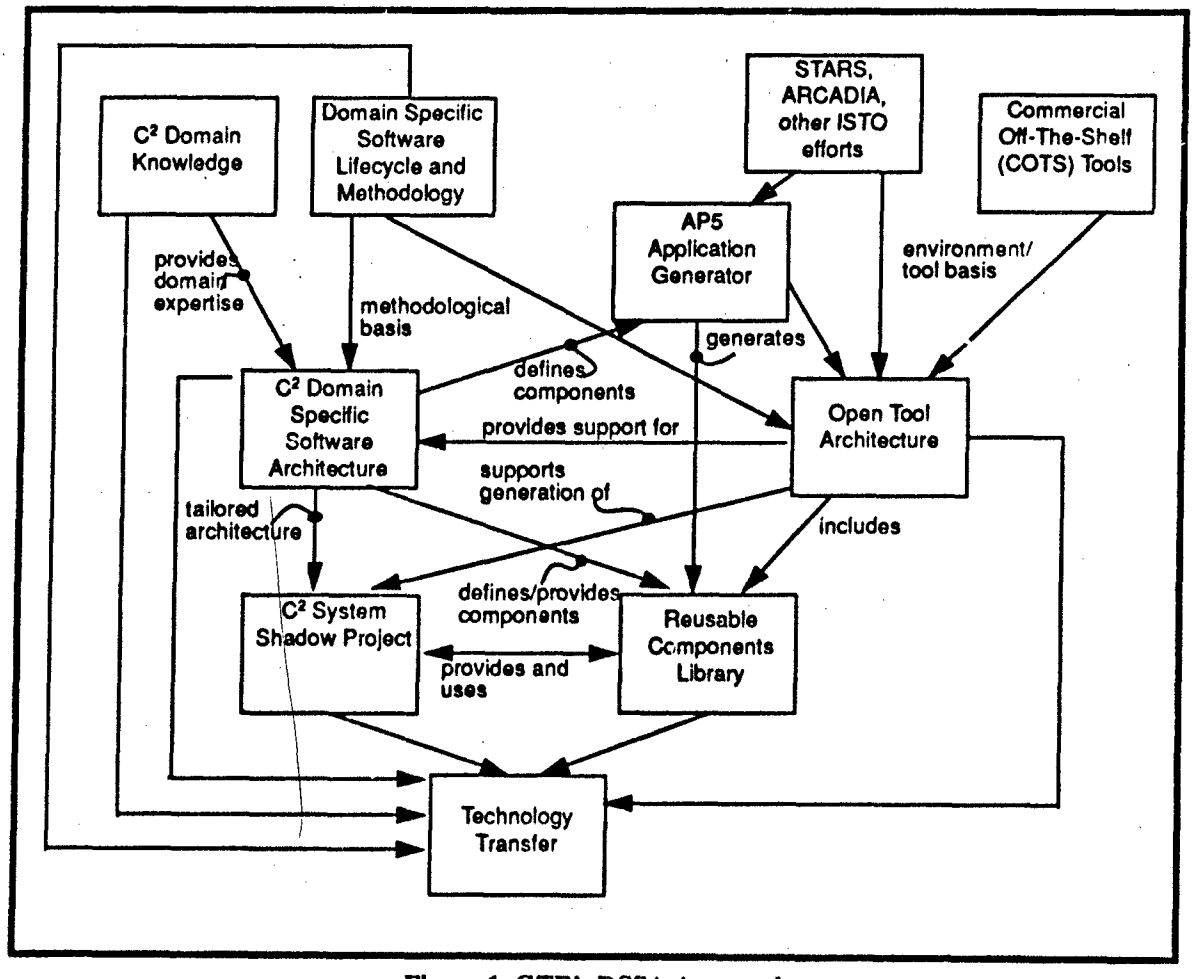


Figure 1. GTE's DSSA Approach

The first is defining a software lifecycle process model appropriate to architecture-driven software development and developing a toolset to support that process. The second is establishing a capability that implements the process for the command and control domain, based on a C2 architecture and a set of reusable C2 components.

The DSSA process model addresses all aspects of the software life cycle. It describes activities for establishing system requirements, developing the software system, and sustaining the system after delivery. The detailed process model identifies roles of government (e.g., PEOs, PMs), developers, maintainers, and reuse library organizations in an architecture-driven, reuse-based lifecycle.

The DSSA toolset will support all of these activities, automating them as far as possible. In particular, it will automate system development activities by using the architecture as a template, guiding the selection of available reusable components, and automating the generation of specific required components. The toolset will be constructed insofar as possible from available tools—both commercial products and products of the research community. In particular, it will make use of USC/Information Sciences Institute's AP5 application generator, DARPA/STARS reuse libraries, and DARPA/Prototech tools. Open tool interfaces will be emphasized to minimize specific tool dependencies, thus making the toolset usable in the widest range of environments

Fundamental to the C2 DSSA capability is the development of a C2 software architecture. This starts with development of a multi-viewpoint domain model, created through interaction with all elements of the DoD C2 community. Tools/methods used in modeling include IDEF0, Requirements Driven Development (RDD), and OMT object modeling. From this set of models, an object-oriented software architecture is being developed. The architecture will tie back to the multi-viewpoint model so that mappings to different views of the domain functional decomposition are apparent. (George Mason University's Center for C3I is playing a major part in this modeling and consensus-building activity.) A base of components conforming to the architecture will then be developed. Many of these will be existing components, perhaps modified to fit the architecture. Others will be automatically generated using AP5.

The DSSA capability will be demonstrated by development of a prototype C2 subsystem, most likely from the fire support area. An independent metrics/validation task will assess the effectiveness of the approach and gather metrics. The methodology and toolset will be revised based on findings and further necessary research will be identified.

Throughout the program, a technology transfer task will present results in conferences, papers, seminars, and short courses. The George Mason University Center for C3I will serve as a focal point for technology transfer.

DSSA Lifecycle Overview

Figure 2 presents an overview of the DSSA lifecycle. The shaded boxes in the figure represent *architecture development* activities and products. These establish the basis for subsequent development of specific applications in the domain. The *domain reference requirements* define the functional and performance requirements that characterize systems in the domain. These are then mapped to a *reference architecture* for the domain—a generic architecture that can be adapted to build specific systems.

Components that conform to the reference architecture are then developed and/or acquired and cataloged in a reusable component library.

The clear boxes in the figure represent the activities and products of *target system generation.* These are the activities in which a developer of a specific system makes use of the architecture products to construct that system. First, through a process of *requirements elicitation*, the target system requirements are analyzed and expressed in terms of the reference requirements model. Then, based on this correspondence, the reference architecture is *instantiated* (adapted, filled in, modified as necessary) to create a design architecture for the specific system to be developed. Components from the library are then used to realize the design—i.e., to create the target system implementation. The design, as it is based on the same architecture that formed the basis for the component collection, largely automates the component identification/selection process.

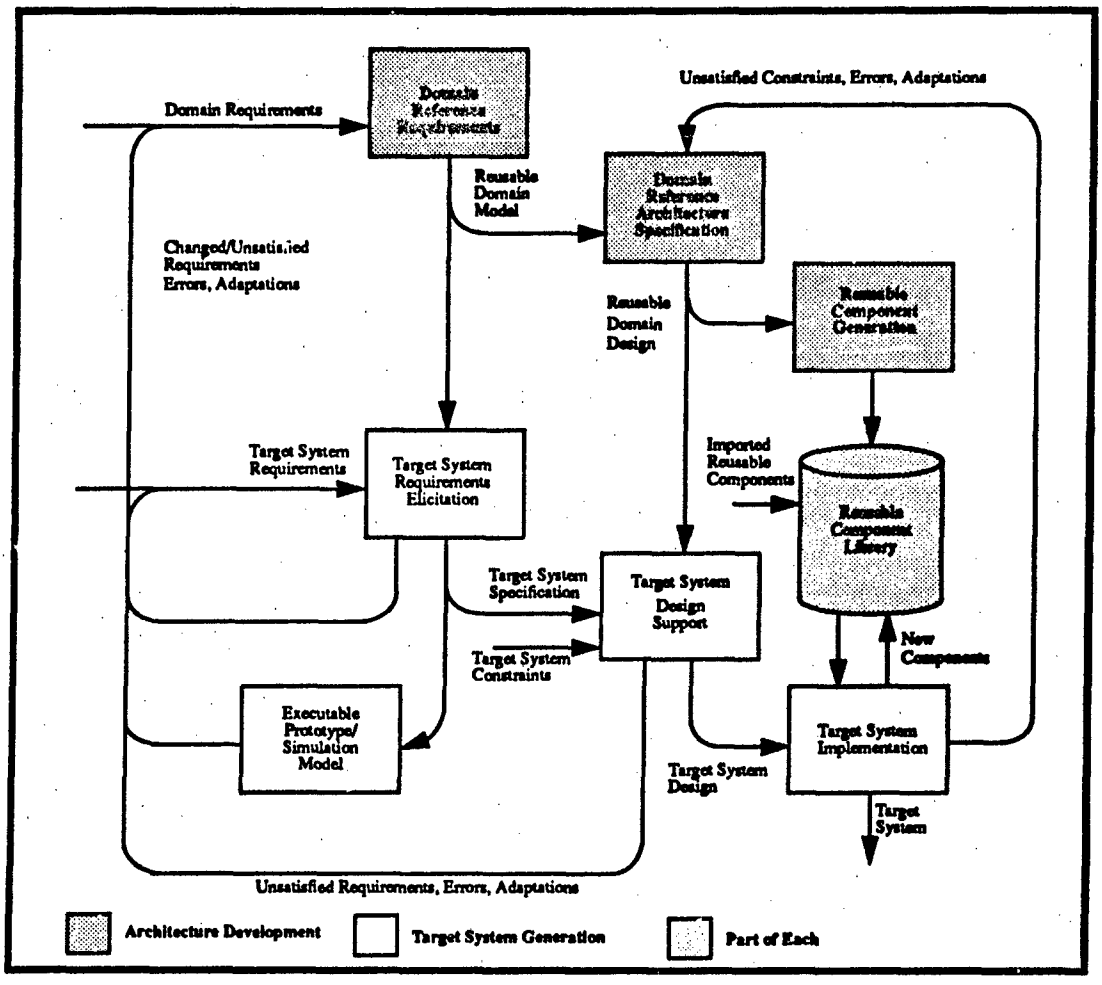


Figure 2. The DSSA Life Cycle Model

The DSSA framework also establishes a natural basis for construction of executable prototypes and simulations. Such prototypes can be constructed from the reference architecture and library components in order to help refine requirements and assess performance. The following sections describe DSSA roles and processes in more detail.

DSSA Roles and Agents

As part of understanding the DSSA development process, it is important to understand the kinds of individuals and organizations that participate in the process. In this characterization, we adopted descriptive terminology used by the Software Engineering Institute's process description project. This approach begins by defining process *roles* and *agents*.

DSSA Roles

A *role* is a uniquely-identified class of individuals based on qualification, skills, or responsibilities that perform specific activities in the process. In addition to the traditional software development roles, the DSSA process defines two new roles:

The Domain Expert. The domain expert is an individual who has wide experience in applications

that distinguish this domain. To be considered an "expert" for this domain, the individual should be able to express application requirements in the framework of a modeling technique intrinsic to the DSSA development environment. In addition, the requirements that are most critical for the domain expert are those that reflect a complete understanding of end user needs. Evaluation of design decisions by the domain expert should reflect the user's perspective.

Domain experts help define and model the reference requirements and continue to be a resource for evaluation and enhancement of applications. They are in communication with and represent the needs of the end user. Example domain experts might be military requirements developers in organizations such as TRADOC. In the commercial world, they might be people in a vendor's marketing organization who specify what the market wants and set the direction for future products.

An experienced software engineer would not be considered a domain expert on the strength of development of applications in the domain alone; he or she would need expert knowledge of the end user's future as well as present needs. Domain experts most likely would have been end users themselves at one time.

The Domain Architect. The domain architect is a system/software engineer who has significant experience developing applications in this domain but does not necessarily have the domain expert's understanding of user needs. This individual must have in-depth knowledge of all aspects of the DSSA process and products. Together with the domain expert, the domain architect elicits and models domain requirements. In addition, it is his or her task to transform the reference requirements to a reference architecture for the DSSA environment. Besides the models of the reference requirements, the assets available to the domain architect would include: consensus models, component class specifications, and reusable components.

The domain architect's products are continually evolving through feedback, evaluation, and restructuring of the environment in addition to the normal system maintenance functions. Maintenance is much more complex in DSSA than in traditional software procedures, as we will note later.

DSSA Agents

An *agent* is an entity that enacts or participates in a DSSA process. An agent may be an organization or a designated role within an organization.

Domain Manager. The organization managing a family of related systems within a DSSA domain is referred to as the domain manager. This is the organization that will gain directly from the common technology base of a domain-specific software architecture. For example, a military Program Executive Office (PEO) might be a domain manager. In industry, a business are or program office manager might be a domain manager. Additional responsibilities are to provide direction to program managers, control budgets and schedules, and set strategic direction for the evolution and use of the DSSA environment. The domain architect's role is completely incorporated in this organization. The domain manager also enlists the aid of a domain expert.

Application Developer. An application developer may be a contractor, vendor, or government organization that develops new application systems. This organization must practice a software development process based on the reference model, reference architecture, and library components. With the help of a domain expert, the application developer builds target systems that are extensions, tailoring, or modifications of the reference requirements and architecture.

End User. An end user is an organization that uses the system built by the DSSA process. Although the end user need not have knowledge of the DSSA development environment per se, he or she provides important information as to the content of domain knowledge, new and changing requirements, adequacy of documentation, and system effectiveness. User needs are dynamic; static systems become obsolete. Therefore, a domain expert constantly revises his or her domain knowledge, and end users are a key input to this. Not only applications but reference requirements and architectures may need to reflect the end user inputs.

Library Center. The library center is an organization responsible for acquiring and maintaining the

domain-specific components and managing the library. The library center may be in the direct control of the domain manager organization or it may be an independent, external organization (for example, a STARS or CIM library). The task of this organization is to provide access to an organized collection of reusable software components. It classifies and installs components, performs configuration management, and collects usage metrics. It develops strategies for component acquisition and provides user services. The library center also coordinates maintenance and enhancement of components, whether in response to specific application needs or to reflect revisions in reference requirements and architecture.

Maintenance Center. A maintenance center is an organization that changes and improves fielded systems. For example, a Post Deployment Support Center (PDSS) is a maintenance center. The DSSA approach increases the operational complexity of this type of organization. Components requiring maintenance may not "belong" to the center, but may come from a reuse library. Furthermore, necessary changes to reference models and architectures may be identified. Accounting for the possible ripple effect of any particular change to the family of systems requires analysis. Areas of requirements traceability, testing, and redesign may not follow traditional procedures. Maintenance is an important consideration in the design of the DSSA process and development environment.

The DSSA Process

We have developed a detailed IDEF0 model of the DSSA process. The top level diagram of that model (Figure 3) identifies the major four phases of the DSSA life cycle. A description of each phase, with its major process steps, follows.

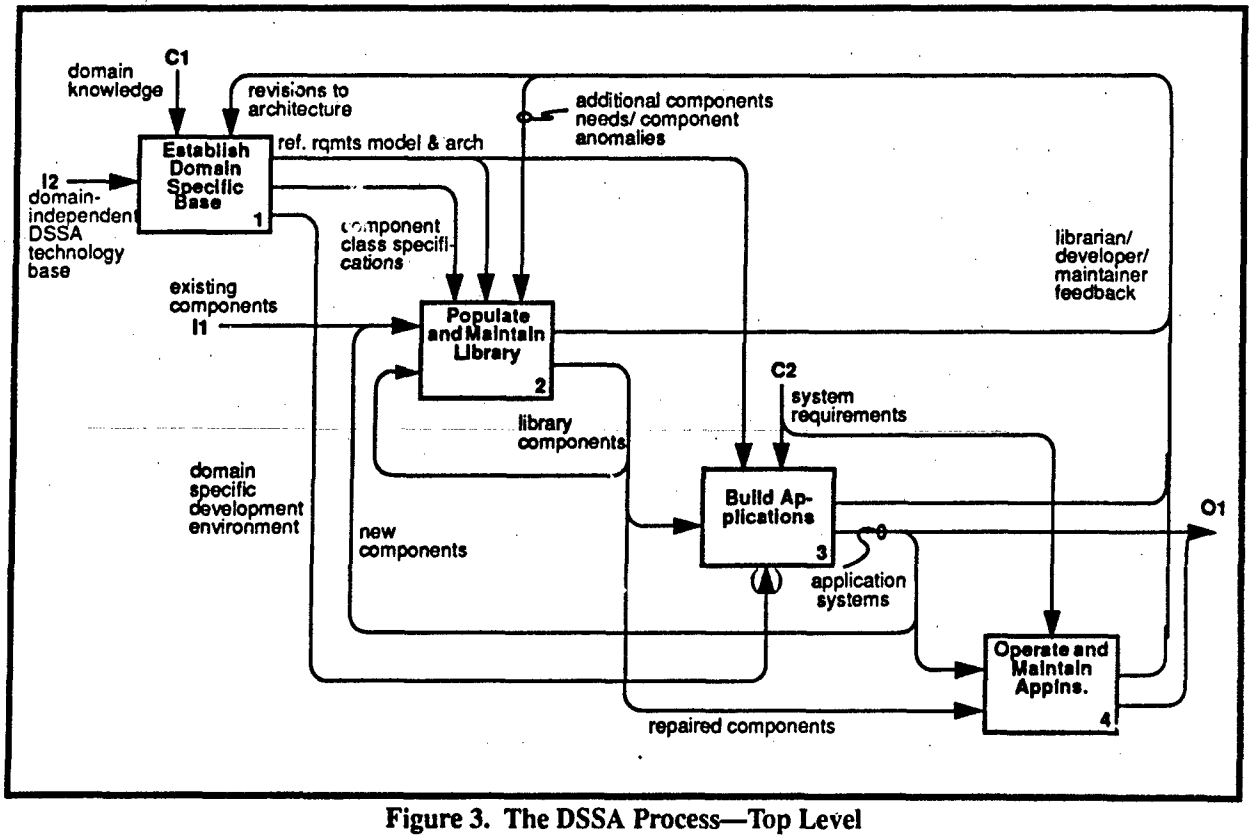


Figure 3. The DSSA Process—Top Level

Establish Domain-Specific Base

This is the set of activities performed by the domain owner to establish a DSSA capability base for the domain. The domain is analyzed and a domain-specific software architecture (reference architecture) and development environment are provided. Specific activities are:

Model Multiple Views. Multiple views of the domain are provided by domain experts to ensure that the broadest range of intrinsic concepts is addressed These multiple views may reflect different types of systems within the domain (for example, strategic vs. tactical C2 systems), or may reflect different perspectives on the domain (for example, that of the soldier in the field vs. that of the strategic planner). These views are expressed in a set of domain models, developed using a multiparadigm combination of object-oriented, dynamic, and functional descriptions.

Establish Consensus Model. A consensus of the generic domain requirements is developed to build a reference requirements model, identifying the functional and performance characteristics of the family of systems. This is a model of the reference requirements that will be the foundation for a family of DSSA systems; it is one of the key DSSA products, playing a key role in application development. This activity also establishes common terminology to describe elements of the domain. Such consensus building will draw on the multiple view models, but will also employ workshops or similar interactions to help build agreement.

Allocate Requirements to Reference Architecture. The reference requirements are allocated to architectural elements (i.e., system objects—software or hardware) to create a reference architecture for the domain. The reference architecture identifies objects, their interfaces, and their topology. It serves as a basis for design of specific systems in the domain. The architecture will continually evolve as new needs arise from various sources such as the application developer or the maintenance center.

Specify Reusable Component Classes. The architecture defines classes of software objects (for example, *forms manager, message handler*) that can be used to build applications following the architecture. Multiple components can be provided to implement each class, varying perhaps in functional, performance, or platform particulars, but to work with the architecture each must conform to the a specification for that class. The specification will identify all interfaces and behavioral characteristics on which the rest of the architecture depends.

Tailor Environment to Domain. The DSSA process assumes the existence of a DSSA development environment, i.e., a set of tools that support the process steps described here. The base toolset is domain independent—it can be used to support a DSSA process for any domain—but several of the tools are tailorable or parameterizable based on the reference requirements and architecture. This step creates these tailored instances of the tools for the domain being addressed.

Populate and Maintain Library

This is the set of activities that create and administer the collection of reusable components that implement the DSSA. Components meeting the component class specifications in the reference architecture are collected, modified, and/or developed. Specific activities are:

Develop Acquisition Strategy. The domain architect identifies sources that will provide one or more components meeting the component class specifications. An acquisition strategy will be developed to identify components and acquire them. Approaches may include:

- use as-is existing components
- reengineer existing components
- build an application generator
- develop components manually

Provide Components. The acquisition strategy is implemented to provide components conforming to the architecture. If problems arise in doing this—for example, ambiguities in the class specifications—feedback must be provided to the domain owner.

Install in DSSA Library. Developed components are then installed in the library supporting the domain. Depending on whether the library is "owned" by the domain owner or is an independent "public" library, this interface may differ. If a public library is used, the domain owner must be proactive in ensuring that needed components are added to the library and continue to be available, at the same time conforming to any entrance requirements the library might have. It is important to note that library components are not necessarily code; typically design, documentation, and

test components will also be provided.

Build Applications

This is the set of activities required to build a specific application using the DSSA library and the domain-specific development environment. Specific activities are defined below. (Note that, while names are like those in the traditional waterfall model, activities are quire different.) The DSSA environment will provide comprehensive support for this phase, incorporating an intelligent decision support capability that automates each activity, additionally providing a requirements traceability mechanism and supporting prototyping, evaluation (metrics, etc.) and testing.

At any step in this process, any deficiencies or needed improvements in the reference requirements, architecture, or components are fed back to responsible organizations (the domain owner or the library manager).

Develop Requirements. In this step the reference requirements model is used as a basis for stating the requirements for the specific system to be built (the target system). The target system's requirements are expressed in terms of the reference model, for example:

- parameterized requirements, e.g., performance measures, are supplied with target system values
- selections among alternative capabilities are made
- unneeded capabilities are eliminated
- additional detail is supplied as needed

The DSSA tools will support this requirements elicitation process, interacting with the designer to create the target system requirements. Rapid prototyping can easily and naturally be included to help clarify requirement distinctions.

Design Application System. The reference architecture is then instantiated and particularized to establish a design architecture for the target system. This process is guided by the mapping from the reference requirements to the target system requirements, and in fact occurs concurrently with that activity. As each requirement is specialized for the target system, the corresponding element of the reference architecture is adapted to become a part of the target system architecture. Design components in the DSSA library are extracted to build the design.

Implement System.

Creation of the application system implementation (the actual code) follows naturally and automatically from the preceding steps. As each design element is chosen, a corresponding library code component is identified. Where the developer must provide adaptation or "glue" software to connect these components, the tools identify this need and provide the needed interface specifications. If the toolset provides a component generator for the particular component class, the developer is automatically guided to its user interface, and it is designed to generate only components that conform to the architecture.

Operate and Maintain Applications

Once built, the application system is operated in the field and maintained by the maintenance center. Specific activities are:

Carry Out Application. The application system is deployed and operated in the field.

Assess Effectiveness. As part of its normal operation, the effectiveness of the system is assessed against the needs of its continually changing mission. Needed changes and corrections are identified and reported to the maintenance center.

Maintain System. The maintenance center responds to all requests from users for changes or corrections to the system. This is their traditional role; however the DSSA approach requires a much different approach to system maintenance than when systems are built independently. An enhancement or correction may affect documentation, system requirements, library components, and eventually reference requirements and architecture. Some of these types of components may not be under maintenance control of the maintenance center. For example, library components may be maintained by the library organization, and the reference requirements and architecture may be maintained by the domain owner. Further, changes to these entities will potentially impact more than the one application system that requested the change. A close working relationship

between the domain owner and the maintenance center is critical to the success of this activity.

Implications and Future Directions

DSSA promises a real revolution in the way we build software. However, as this brief overview indicates, the DSSA process will impact significantly the roles and responsibilities of all organizations involved in the creation and support of software. This creates a challenge to the industry overall, but meeting this challenge must start with the DSSA teams. It is essential that we develop a process that is:

- clearly defined and described
- comprehensive—addresses all elements impacted by the change
- practically realizable with today's resources
- supported by tools
- accepted by participants in the process

The process model described in this paper (and in more detail in the referenced materials) is a beginning at meeting these needs, but it is only a beginning. The model will evolve as we gain experience on the project as the first practitioners of DSSA, and as we explore the limits of what is possible in tooling to support the process.

Acknowledgment

The work described in this paper has been supported by the Defense Advance Research Projects Agency through U.S. Army Communications-Electronics Command Contract No. DAAB07-92-C-Q502.

References

[1] Armitage, James, "Process Guide for the DSSA Process Life Cycle," PD-081 DSSA-PG-001 Rev. 0.1, Software Engineering Institute, 10/92.

[2] Braun, Christine, Raymond Coutant, and Jorge Rodriguez, "DSSA Process Model", GTE working paper, 1992.

[3] Braun, Christine, W. Hatch, T. Ruegsegger, B. Balzer, M. Feather, N. Goldman, and D. Wile, "Domain Specific Software Architectures— Command and Control", *Proceedings of 1992 IEEE Symposium on Computer-Aided Control System Design*, Napa, CA, March 1992.

DOMAIN ENGINEERING:

Establishing Large-Scale, Systematic Software Reuse

William R. Stewart **William G. Vitaletti**

SofTech, Inc.
Alexandria, VA

Abstract: Domain Engineering is a collection of activities (Domain Identification, Domain Analysis, Domain Design, Domain Implementation) that provide generic requirements and designs for a given domain (family of systems or common system service). These, in turn, are tailorable to a particular system based on differing factors such as mission, site, environment, new technology, and user needs. The products ideally contain the knowledge base of the domain and include reuse guidance incorporating rationale, alternatives discarded, and lessons learned.

Keywords: Domain analysis, domain design, reuse, systematic, opportunistic, components, object-oriented, commonalities, adaptation, generic requirements, generic design, DSSA, and generic architecture.

1 INTRODUCTION

Software reuse is widely recognized by industry and government alike as a primary mechanism to combat today's software crisis. Many experts insist that software reuse is an effective means to increase software development and maintenance productivity, leads to greater quality and more reliable software, and can preserve software engineering expertise[1]. Although increased productivity is often cited as a key reason to practice software reuse, the most significant benefits may come from increased reliability and lower software maintenance costs. However, even with the potential to yield higher-level software productivity gains and demonstrated results of several reuse case studies, there still remain many barriers to reuse. To overcome these barriers and maximize benefits, the software reuse process requires planning and methodical integration into the software development life-cycle (SDLC).

Employing a domain analysis and design process is one of several existing approaches to effectively identify commonality and engineer systems for reuse[2]. In general, domain analysis and design is the systematic exploration of related software systems to discover and exploit commonalities; to produce a set of common capabilities, processes and data for a "family" or class of systems; to represent and model commonalities in a usable form; and to provide a method to map commonality to specific reuse instances. The primary objectives of domain analysis and design are to understand the domain, to support user-developer communication, to provide reuse requirements, and to develop products that support implementation of new applications[3]. By applying domain analysis modeling techniques and designing generic architectures, engineering activities focus primarily on developing a set of common requirements and exploiting adaptable architecture(s) for a "family" of systems (hereafter referred to as a *domain*).

1.1 Types of Software Reuse

Software reuse doesn't just happen, it must be integrated into the SDLC. Many software industry leaders have adopted either an *opportunistic* or a *systematic* approach to developing reusable software. Opportunistic (sometimes referred to as ad hoc) reuse has been practiced for years in an unstructured manner by those sharing code modules. The potential

benefits of this software parts-based approach to software engineering are significant, but they are based on assumptions that a given domain exhibits significant commonality. Commonalities are then exploited through the development and application of reusable software components or parts. In this approach, reuse benefits are achieved only at the component or part level.

In order to realize greater productivity gains, which are needed to overcome the software crisis, a systematic approach to reuse is most useful. This type of approach is formalized at more than an ad hoc, code-sharing level. Structured, repeatable methods are employed to focus on emerging technological software engineering activities, such as domain analysis and design, to maximize the benefits of reuse. Domain analysis is a methodical process, or set of activities, used to acquire and model an understanding of the domain and the specifications common to systems within the domain. Domain design activities use the domain model and specifications to develop generic domain designs as configurable or adaptable frameworks for constructing new systems in the domain. Reusable software components that implement domain specifications are then acquired and/or developed for next generation system development. New development efforts in the domain can reuse domain specifications, instantiate domain design(s) and reuse the associated lower-level implementation components, thus realizing far greater benefits of reuse.

1.2 Problem Space vs. Solution Space

The domain problem space is the set of requirements that future systems in the domain will need. The domain solution space represents the implementation of the problem. Successful reuse requires the developer to match a current set of problems (system requirements) to previous solutions. The design and/or implementation of the previous requirement can then be adapted as a solution to the current problem. Domain engineering provides a systematic approach that concentrates specifically on defining problems and producing adaptable solutions in a domain in order to increase productivity in future system development activities[5]. This process, which complements and facilitates system development, is an essential part of an effective, reuse-based development methodology.

1.2.1 Problem Space. In traditional system software development, problems are formulated and defined as requirements for specific software systems (or parts of systems). Domain analysis identifies and models the problem space for a "family" of systems (domain), and provides reusable requirements to address recurring problems in the domain. The problem space defines common capabilities within the domain, as well as associated variations and combinations. It is essential to define domain capabilities to be effective, flexible and remain viable over changes in technology, time, needs, people and budget. To support these changes, problem space requirements, must be adaptable and understandable.

Domain models are developed as abstract representations of systems existing in the domain and/or future domain requirements. These abstract domain representations serve as models for constructing future systems. The interpretation of domain models is flexible and allows alternative realizations to accomplish different needs.

1.2.2 Solution Space. Domain design focuses on the solution space where solutions take the form of design and other artifacts that together constitute a framework to address the problems in the domain. Domain design applies system problem solving to construct a generic, adaptable design that satisfies domain requirements. This generic design serves as the basis for the system software design activities. During system software design, a software architecture that exploits the implementation dependencies of the target environment is constructed.

Domain architectures allow the developer to create robust language-independent solutions satisfying those requirements, without concern for how the software will be implemented. This preliminary, logical design is the developmental bridge between the problem domain and the software implementation or solution space. The resulting problem domain and solution space mapping is essential to effectively maximize the benefits of software reuse.

1.2.3 Increased Benefits. There is potentially a much greater return on investment by employing a systematic reuse-based approach and

applying the domain analysis and generic architecture design techniques described in this paper. For instance, if requirements for developing a new system match (to a certain degree) an existing domain model, it may be more feasible to adapt the corresponding domain design and reuse large portions of associated existing system components, such as detailed design(s) and code.

This paper presents one method of domain modeling and generic architecture design, and includes essential activities in conducting the domain analysis and design process to enable software reuse and maximize the associated benefits.

2.0 DOMAIN ENGINEERING LIFE-CYCLE

Domain engineering is the process by which all domain products are created. The four major activities of domain engineering are:

1. Domain Identification
2. Domain Analysis
3. Domain Design
4. Domain Implementation*

As in standard software development, various life-cycles are possible, from waterfall to spiral to incremental.

The products of domain engineering relate to the application life-cycle as shown in Figure 1. The domain models, from which the generic requirements are generated, are composed of graphical representations and object specifications. The architectures are composed of generic designs. After completion, the domain products are placed in reuse repositories for accessibility.

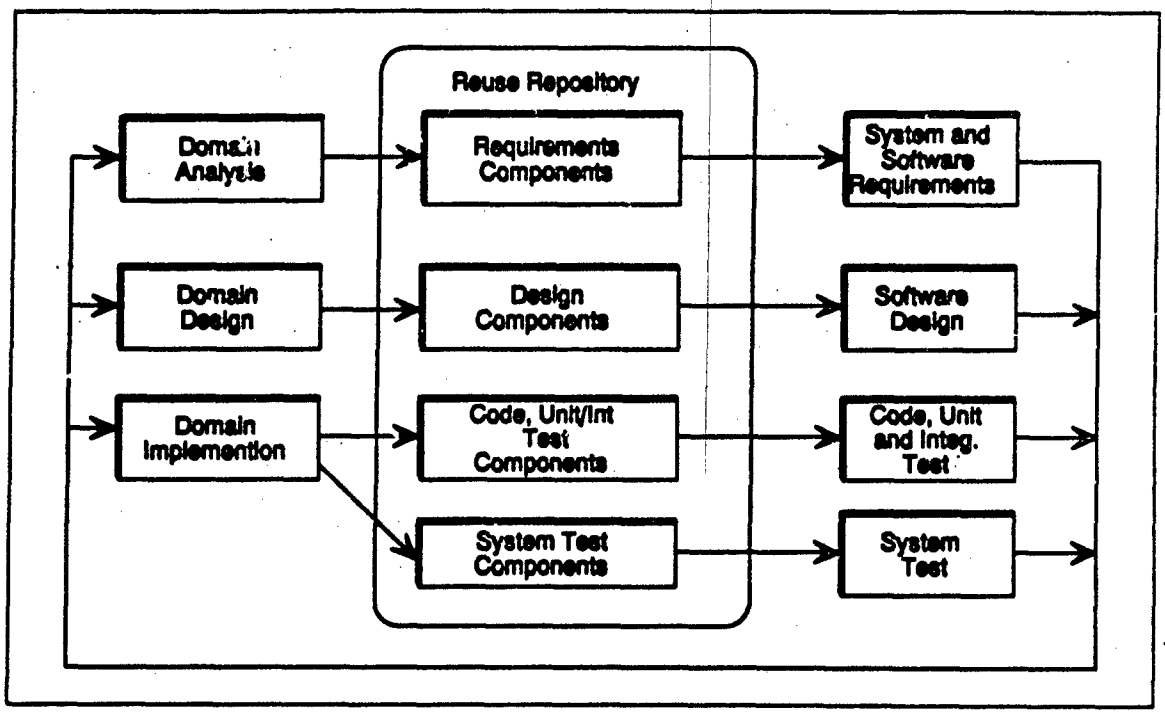


Figure 1. Integration of Domain Engineering and Application Engineering Life-Cycles.

* Domain implementation is basically the application of well-known coding and/or re-engineering principles and, therefore, will not be covered within this paper.

2.1 Method Overview

As in the practice of standard software analysis and design, several domain analysis methods exist at the present time. In order to classify and combine common system/software capabilities, the method presented in this paper is a completely object-oriented approach. This approach was adopted for several reasons:

1. Object-oriented methods provide inherent constructs for identifying commonality (classes), variability (subclasses) and cardinality (instance connections).

2. Traceability is established on an object basis from requirements through code. Every object in the requirements phase has one or more objects in the design and implementation phase.

3. The transition from object-oriented analysis (OOA) to object-oriented design is greatly simplified. "Because of the difference in aggregation principles, proceeding from a structured analysis to an object-oriented design can be awkward. Since the criteria for grouping functions are different in the two methods, the transition from one to the other may require significant recasting of the data flow diagrams. This is a laborious process, which can be avoided by assuming an object-oriented viewpoint during the analysis phase".[6]

4. During the construction of an object, both during analysis and design, changes within an object will have little or no ripple effect in other objects, allowing the information clustering to proceed without extensive revisiting of related objects.

Although many existing systems have been developed with functional or hybrid methods, the information clustering necessary to identify commonality requires a complete information restructuring.

The specific graphical techniques currently employed in this process are Coad/Yourdon analysis notation[9] and Buhr '90 design notation[8]. Several other analysis and design notations are presently being considered in light of emerging domain engineering needs.

The detailed examples provided herein to demonstrate concepts were derived from: 1) The U.S. Army Reuse Center (Software Development Center - Washington) Program Executive Office - Standard Army Management Information Systems (PEO-STAMIS) retail supply domain analysis effort; 2) information from other supply systems documentation; and 3) ongoing method enhancements. For this paper, the information has been greatly simplified for purposes of demonstration and should therefore be considered in the light of concept communication *only*.

2.2 Domain Identification

Before an analysis of a domain can begin, a domain must be defined. The first consideration in domain definition is the *orientation* of the domain in question. The domain orientation is either *vertical* or *horizontal*. If the reuse projected involves a family of related systems, it is said to be a vertical domain[13]. If the reuse projected involves a segment of many families of systems, it is said to be a horizontal domain (see Figure 2). A mission-specific domain (e.g., Radar-Guided Missiles or Logistics) tends to be vertical and a system support domain (e.g., COTS bindings, GUIs, Communications, or Data Base interfaces) tends to be horizontal.

In order to understand the extent of the domain engineering effort, the *boundary* must be established. The boundary identifies the characteristics that are used to classify systems as part of a specific domain. As a high-level example, the Patriot Missile System belongs to a Radar Guided Missile domain and the Sidewinder (infra-red guided) does not.

Example:

Under the PEO-STAMIS domain analysis effort, approximately 26 systems were identified as candidate PEO-STAMIS systems. During this effort, the fundamental capabilities of these systems were assessed to identify similarities. Organizational entities were examined to identify responsibilities. The results in Figure 3 portray the domain organization of PEO-STAMIS. The Supply domain was chosen from the domains in PEO-STAMIS in

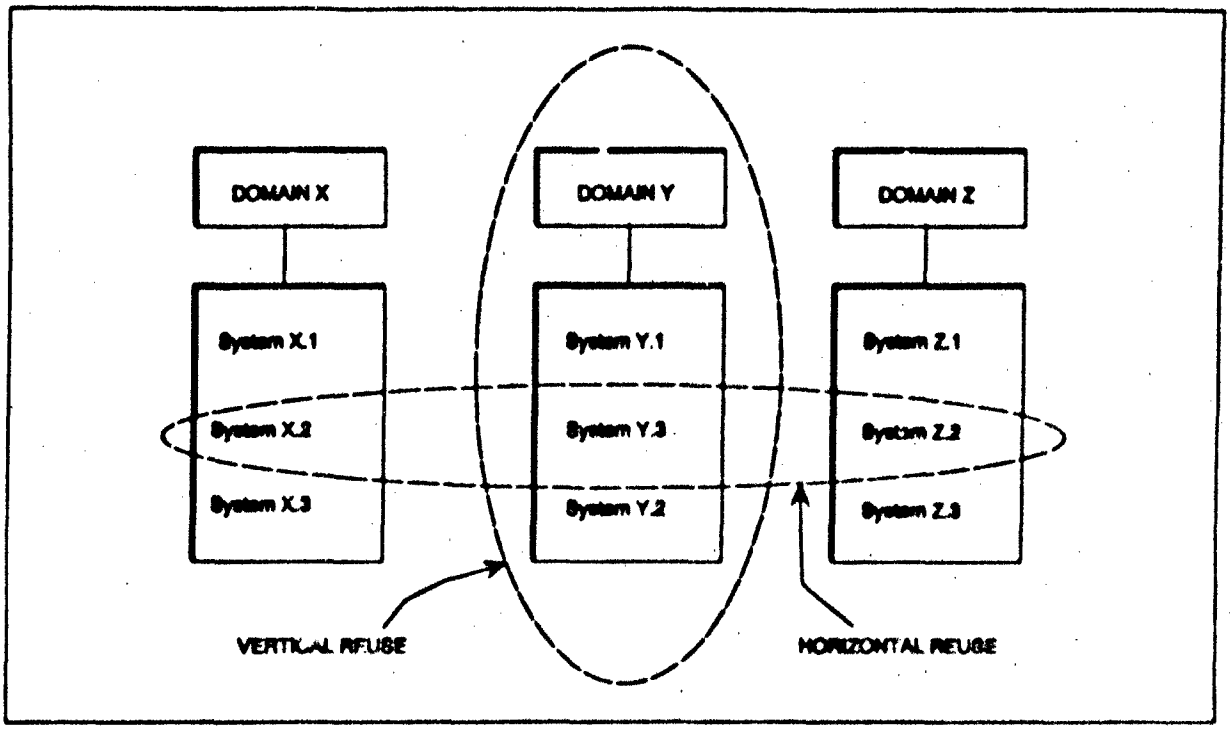


Figure 2. Domain Orientation (Horizontal vs. Vertical)

order to utilize the wealth of available information and to satisfy the needs of upcoming Supply systems.

The reuse potential discovered included both horizontal and vertical reuse. For the sake of brevity, this paper will focus on vertical reuse. The boundaries of the domain analysis were drawn around the systems that had supply responsibilities (see Figure 3).

In general, the PEO-STAMIS Supply Domain consists of logistics systems that provide the fundamental capability to supply customers with the assets required to accomplish their respective missions. Customers and customer needs may vary at different echelon levels within the Army's organizational structure. For systems at the brigade, battalion, company echelon, for example, the customers may be tactical commanders. Whereas, customers at the theater, corps and division echelons may be motor pool mechanics or intermediate supply points.

2.3 Domain Analysis

The objective of domain analysis is to identify, derive, organize, abstract, and represent the body of knowledge of a particular domain[2]. This body of knowledge is represented by a *domain model*, a model similar to an Object-Oriented requirements Analysis (OOA). The difference between a domain model and an OOA are the additions in the domain model of reuse guidance in the form of adaptation requirements, alternatives discarded, rationale, lessons learned, and identification of systems supplying or consuming the domain products. This information is the key to providing the extra information needed to construct requirements for many future systems. The domain model provides input to both the domain designer and the application software analyst.

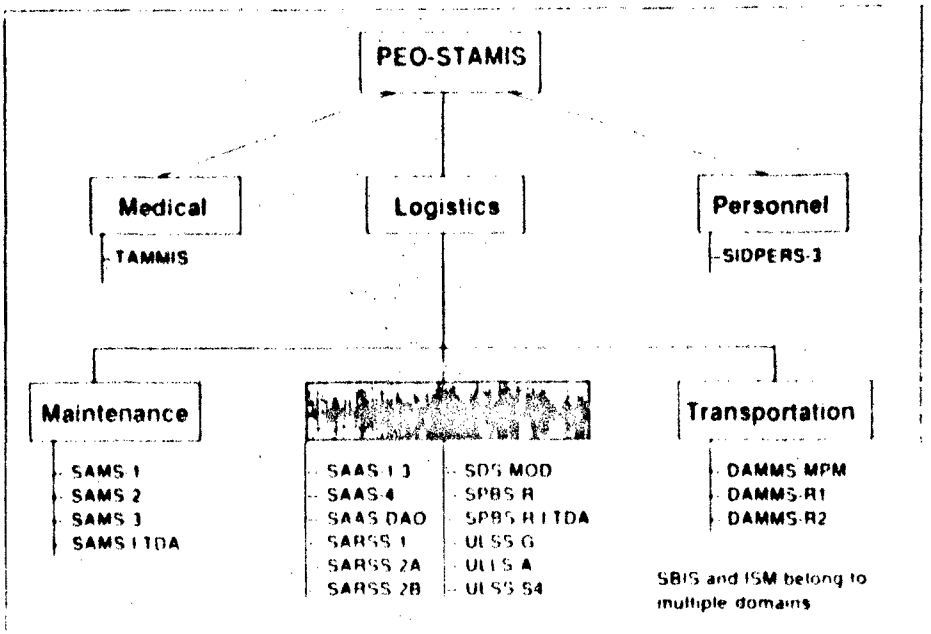


Figure 3. PEO-STAMIS Supply Domain Boundaries.

Different domain analysis methods vary in their goals as to the depth of information to model. For example, the Software Engineering Institute's Feature Oriented Domain Analysis (FODA) aims at capturing user-visible aspects of the problem space as a means of identifying user requirements. Additionally, other methods include all of the problem space aspects in the domain model as means of "disseminating knowledge among engineers as a way of improving the development process". [5]

The primary steps of domain analysis are:

1. Information Gathering and Organization,
2. Commonality Identification,
3. Adaptation Identification, and
4. Domain Model Verification

These steps can be performed in sequence, although experience has shown that iterations can produce greater detail and understanding thereby increasing reuse potential. Information gathering tends to continue into commonality identification, as some documents may reference other publications needed or helpful to the analysis effort. The identification of adaptation requirements often takes place each time a commonality is discovered.

2.3.1 Information Gathering and Organization. During this step, the domain analysts acquire information, at a minimum, in the form of system documents, planning/concept documents, and initial domain expert interviews. This information becomes the basis for a *working domain knowledge base*. This knowledge base will be used throughout the domain analysis and design activities before the completed products are formally certified and installed in the reuse repository. Ideally, all significant information will be included in the certified domain products. The types of information to focus on are those that will have some impact on future systems. Obsolete or out-dated information is not maintained in the domain model, although knowledge of the amount and rate of obsolescence can help determine the stability of a domain.

The knowledge base needs an organizational structure. Rapid access to particular groupings of domain information requires an indexing of the information according to the type of information acquired (e.g., system document vs. interview, obsolete vs. future need).

2.3.2 Commonality Identification. In order to establish a basis for commonality identification, an entity-relationship model is constructed (unless one is available from a business improvement initiative effort). Commonality identification determines, to a large part, the amount of reuse potential to be realized from the domain engineering effort. The common information can be in the form of objects, classes, functions, processes, entities, relationships, data elements, and so forth. What is searched for is not necessarily exact matches, but similarities. Commonality can be identified in a number of ways:

1. *Identification by Domain Experts* - Such as system users, functional proponents, commanders, and other related individuals that have had to work with and understand the problem domain in question. Their knowledge of repeated processes and data manipulation algorithms is extremely valuable. Different types and levels (i.e., skill area, rank) of domain experts have different perspectives that allow multiple perspectives of commonality.

2. *Examination of Analysis Documentation* - All information in the form of software engineering data, such as object diagrams (interaction, inheritance, aggregation, etc), data flow diagrams, state transition diagrams, and data models, are necessary sources for the modeling of the requirements of the domain. As discussed in CAMP 3[11], software components can exist at different levels of abstraction. We may find similarities at the smallest component level, at a Computer Software Component (CSC) level, or at an architectural level. Often, the same information is represented with different formats, terminology, and in different groupings.

3. *Examination of Design Documentation* - Design documentation portraying previous system solutions can provide commonality information through the study of the use of generics, templates, and other reuse mechanisms. Often, a system's analysis products are in a functional decomposition format. If the corresponding design employed OOD methods, identification of problem space objects in the design can also be useful to the domain analysis process. This constitutes a reverse engineering of the design. The analyst must, however, be alert to avoid introducing solution space information into the problem space and remain focused on only the problem space objects found in the design.

As in object-oriented analysis, commonalities are generalized into classes that represent abstractions of their respective set of objects. This process is not strictly top-down (i.e, without regard for existing assets) in nature, nor is it purely bottom-up (i.e., basing analysis only on existing assets). The goal is to satisfy future requirements with as many existing assets as practicable.

The commonality identification progression does not just proceed from the high level of abstraction to the lowest. Instead, there is a sinusoidal "porpoising" effect as new information discovered in the lower levels of detail affect parent classes/objects. There is an iterative shaping of the classes as new similarities are discovered.

2.3.3 Adaptation Identification. Adaptation requirements identify the different application of the common capabilities. Just as commonality identification is necessary to establish a grounds for reuse, adaptation identification is necessary to tailor the information to particular needs. Some adaptation factors[14] are:

1. Flexibility in operation,
2. Mission adaption (needs, threats, etc.),
3. Environment/site adaptation,
4. Platform adaptation,
5. User adaptation, and
6. New technology adaptation.

These factors can be identified concurrently with the commonality identification.

The techniques in representing adaptation can take many forms. The generalization-specialization feature of OOA provides for subclasses where different configurations can be further represented. The object specifications have adaptation sections in the class/object, attribute, and operation sections. State transition and/or structured logic representations can also identify tailorable needs by modified notation.

Often, another perspective on commonality may be realized in this step, causing a refinement to a class content or hierarchy.

In some cases, certain capabilities may be mutually exclusive and cannot exist together in the same system. This cardinality can be represented by an instance connection that denotes a zero-to-zero relationship between objects/classes.

Example:

The external interfaces to the supply domain are generalized, as shown in Figure 4.

The supply domain has several common characteristics in the form of attributes (data) and operations (functions). Each system in the PEO-STAMIS domain (or nearly all) possessed all of the capabilities listed in Table I.

Next, patterns of variations of those capabilities were sought to identify adaptation requirements. Some of the differences were attributed to the unique requirements in the handling of different supply classes (bullets versus bread, for example). Others were identified by organizational needs (in this case, Army echelons). For example, commanders at the theater level need visibility and modifiability of the distribution of stock to the fielded divisions, whereas unit commanders only need information concerning their own company inventory.

These groupings led to the formation of subclasses representing the variations in system responsibilities (see Figure 5).

2.3.4 Domain Model Verification. The domain model must be verified in order to establish a reliable baseline. The business process and data models, if available, should be used as the "doctrine" against an operational scenario. Verification can be accomplished in the following ways:

1. *Inspection* - A team of domain experts examines the model to verify completeness and correctness. Preferably, at least some of the domain experts will not have taken part in the information gathering, therefore, providing an independent review. Operational scenarios can be used at this step to assure the correctness of the model.

2. *Prototyping* - A small set of capabilities can be implemented using a prototyping tool/language. The prototype would then be executed.

3. *Simulation* - A simulation model can be constructed to determine (to a factor of confidence) the viability of the allocation of performance requirements.

2.4 Design Domain

The goal of domain design is to produce a Domain Specific Software Architecture (DSSA) with reuse guidelines and classification. A DSSA is "based on the concept of an accepted generic architecture for the target domain. As defined by DSSA, a software architecture describes the topology of software components, specifies the component interfaces, and identifies computational models associated with those components."[6] Quanrud has stated, "A generic architecture provides a high level design for a family of related applications and a set of reusable components that are specifically intended for use in those applications. The reusable components are designed to work together and should provide most of the code that would be included in a typical application. Actual applications are developed by adding application specific components and adapting the reusable components to meet the requirements of the application. Adaptation of a reusable component may take the form of modification, extension, use-as-is, or replacement." prescribe a specific completion point for the DSSA, but provides for the tailoring of the level 12.

The level of detail in a DSSA, at the time of this

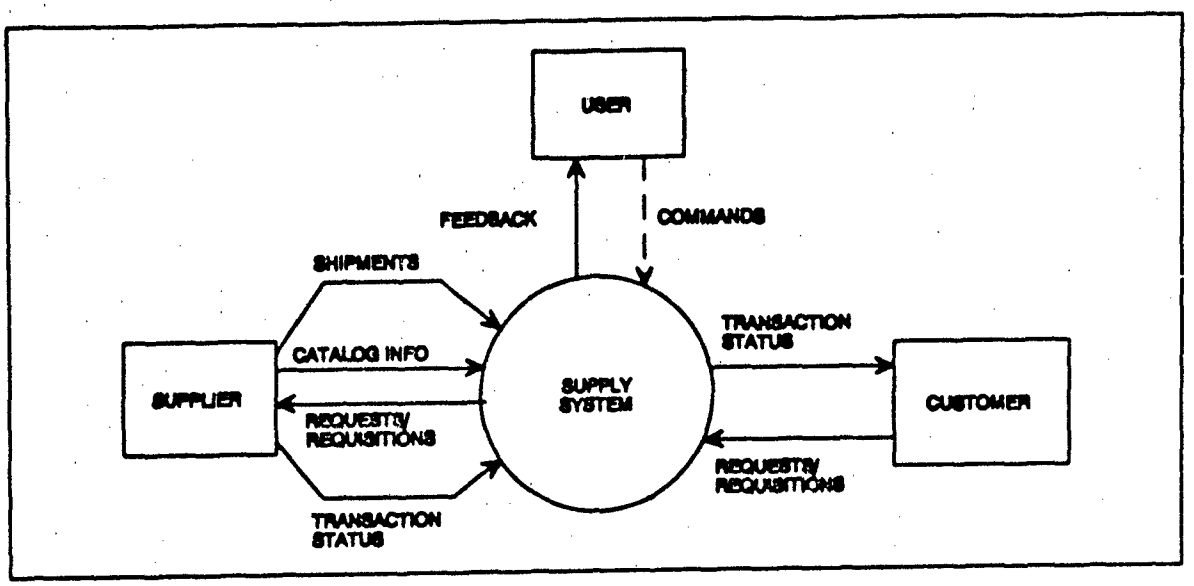


Figure 4. Supply System External Interfaces

Table I. Common Supply Operations and Attributes (Example)

ATTRIBUTES
Transactions Stock Quantities Catalog
OPERATIONS
Accept Incoming Request (for Stock) Request Transaction Receipt/Status Provide Transaction Receipt/Status Cancel/Modify Incoming Request Issue Stock Receive Stock Perform Inventory Maintain Stockage Levels Receive/Update Catalog Disseminate Catalog

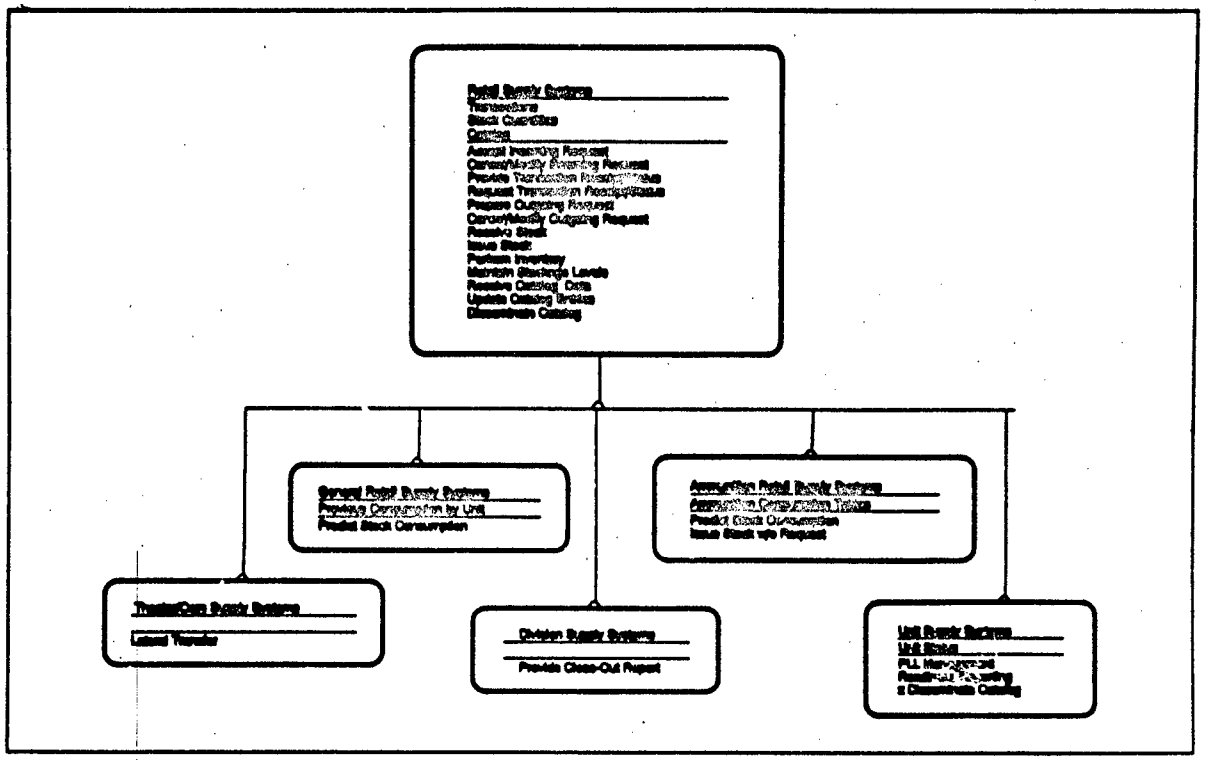


Figure 5. Variations Depicted as Subclasses

writing, has been nominally equivalent to a high-level design. This DISA/CIM Domain Analysis and Design Process does notof detail by providing a completion criteria control to be determined by management.

The purpose of the DSSA is to provide a framework illustrating the major components and their interfaces that satisfy the requirements of the domain model, from commonality to adaptation. This includes reuse guidelines, rationale, and discarded alternatives, in order to give the software designer an understanding of the avenues already explored. The reuse guidelines help the system designer understand the assembly and tailoring of an application from a DSSA.

As in standard software design, the qualities of reusability, adaptability, and efficiency must be balanced to meet the demands of the domain under development. For example, in a "hard" real-time environment, some aspects of reusability and tailorability may have to take secondary consideration to efficiency in order to meet performance requirements[13]. Other domains, especially Management Information Systems (MIS), can realize the cost-saving benefits of more thorough abstraction, encapsulation, and parameterization due to less stringent timing and sizing constraints.

The domain design process is outlined in the following major steps:

1. Identify Domain Constraints,
2. Collect and Organize Design Information,
3. Identify Potential Design Components,
4. Develop Architecture, and
5. Validate Architecture.

The domain design process can be top-down only, bottom-up only, or a combination of the two. The third alternative will be presented herein, although optional processes can be deleted to enable top-down only.

2.4.1 Identify Domain Constraints. The goal of this process is to identify all established and potential constraints affecting the design process. A number of factors, such as imposed standards, Commercial-Off-The-Shelf Software (COTS), Government-Off-The-Shelf Software (GOTS), and hardware, can limit the choices a domain designer can make. A successful domain design meets as many constraints as is practicable, thereby enabling reuse.

The domain designer identifies constraining standards, policies, directives, guidelines, and any other documents that impose design guidance on the domain design. These include decisions as to environments, COTS/GOTS assets, data standardization, and other aspects.

The domain designer researches preplanned hardware constraints to determine if the design:

1. Must be tailored to meet specific goals dictated by timing, sizing requirements, or
2. Will be released from previously established constraints.

The domain designer identifies the domain *system* boundaries as directed by management, that is, the interfaces to other layers/components in the system. For example, management guidance might dictate that a communications domain be implemented without regard to any particular mission-specific domain.

2.4.2 Collect and Organize Design Information. The domain designer creates a catalogue of existing design solutions, domain constraints, and lessons learned. The catalog is needed to provide an index for efficiently accessing pertinent design information without requiring sifting through stacks of documents or hundreds of files.

Existing design information takes many forms. Some projects have complete and up-to-date design documents, while other projects choose not to maintain or keep their documents. A potential source of design information can be acquired by reverse-engineering code, though the algorithmic information is usually difficult to obtain. Other sources include repositories, trade journals and publications, text books, and academic publications.

2.4.3 Identify Potential Design Components. This crucial step enables construction of an architecture, which is the next step, by identifying existing quality design components. This takes place before the actual start of architectural construction so that available components are used as the basis for the design. In the past, designers would often develop a design, then look, often unsuccessfully, for components that fit the design. NOTE: This step is optional if the domain engineering effort is top-down only.

Domain designers and implementors identify components meeting, at least partially, the requirements of the domain model. This provides for reusability of existing components by using them to define the design. Ideally, the components have been developed using similar domain analysis method paradigms to enable an uncomplicated comparison task. For example, if the components were developed using object-oriented design and the domain analysis method is object-oriented, then the information restructuring effort will be minimal. If the domain analysis method is functional in nature, considerable effort will be required to convert one information set into the representation of the other.

The engineers assess the reusability by applying established reusability criteria. This helps to filter out components that will have little or no value in this effort, and to focus on the most promising ones. A number of metrics are captured and the amount of re-engineering is estimated. Based on these results and their comparison to the reuse criteria, a determination is made to either accept the component as a potential candidate, to reject the component, or to decompose the component to allow a more complete assessment.

2.4.4 Develop Architecture. This process utilizes components identified in the previous activity to construct the DSSA. Often, several alternatives can be derived from existing systems that

satisfy the requirements and constraints. If appropriate, one DSSA should be selected that provides the best solution. In very large domains, one overall DSSA may not provide enough commonality to be of substantial use. If for some reason, existing architectures are not used/available, then the standard software design principles are used to create the DSSA from the domain model, within the identified domain constraints. Care must be taken to adhere strongly to the software engineering rule of abstracting detail to the lowest level and isolating implementation dependencies.

This process produces a design specification providing enough information to advance to the next level of design. Criteria established at the beginning of the effort determines the level of detail to be achieved. Until the required detail is reached, the design is continually decomposed and refined.

As in the domain model, reuse guidance is provided to help the system developer tailor the design to the specific needs of the system under development. The domain designer creates a set of guidelines for using the DSSA in a full scale software development activity. These include a discussion of the rational for the selection of a particular alternative, when to use particular components, how/where these components have been used previously, and any lessons learned.

2.4.5 Validate Architecture. To ensure the DSSA created meets the domain model requirements, domain constraints, and provides a workable solution, the domain designers and implementors must validate the DSSA. To do this, alternative approaches exist:

1. *Inspection* - Domain technical experts review the architecture and provide input regarding the correctness, from their perspective.

2. *Simulation* - The software characteristics are used to produce a simulation model focusing on measurable performance.

3. *Prototype* - A small demonstration program is assembled to ensure that the design can be successfully instantiated by an application. Further lessons learned and reuse guidance are captured in this phase.

Example

This example demonstrates the construction of a mission-specific (supply) DSSA. The data and operations listed in Table 1 were grouped together into distinct objects by a closer examination of their content and purpose. In order to manage complexity, the construction of the DSSA proceeded by:

1. Object identification,

2. Object interaction, and

3. Object structure identification.

The first two steps are illustrated by an abstract structure chart (see Figure 5). The third step is illustrated with a concrete structure chart (see Figure 6).

Object Identification - The first data item (TRANSACTION) and the first four operations center around the management of transaction information (a REQUEST is one form of a transaction). This suggests an elaboration of an abstract object "TRANSACTION" as shown in Figure 5. The next data item and the next four operations focus on management of the inventory, and can establish an abstract object "INVENTORY". Finally, the remaining data item and operations manipulate the catalog information used to order stock from the supplier. These are grouped to form the abstract object "CATALOG ".

Object Interaction - The establishment of the interfaces with the external entities (the customer and the supplier) reveal that the only two "visible" objects are "TRANSACTION" and "CATALOG". The "INVENTORY" object receives commands only after the initial transactions are validated, and generates an event to create a transaction if stock levels drop below a preset point. This interface organization allows for an orderly control of the inventory and separates the processing of two types of data (inventory and transaction). The CATALOG object does not need to interface to the other objects as it is only accessed to update the catalog and to order

shipments of stock.

Object Structure Identification - At this stage, the details of the objects, are identified to provide a complete design description. The operations and data that where grouped before to identify objects are now represented along with any structural decomposition. For example, Figure 6 shows the structure of the three objects that include the operations and data from the domain model. The INVENTORY object not only has the Issue Stock, Receive Stock, and Perform Inventory operations, but also a Check Stock Quantities operation needed by TRANSACTIONS to verify stock-on-hand before filling a request. The Maintain Stockage Levels operation is a concurrent process (in this instance an Ada task) that compares the present stock inventory with minimum sustainment levels and calls TRANSACTION to reorder if the minimum is not met. The Stock Quantities data is now decomposed into the three data objects (Stock On-Hand, Stock Due-in, and Stock Due-Out) that are created by instantiating an object constructor (in this instance an Ada type).

Figure 6 shows the calling connections for a scenario where a customer submits a request, the stock is available and is issued, and the action causes a stock item to fall below a minimum level, prompting a reorder.

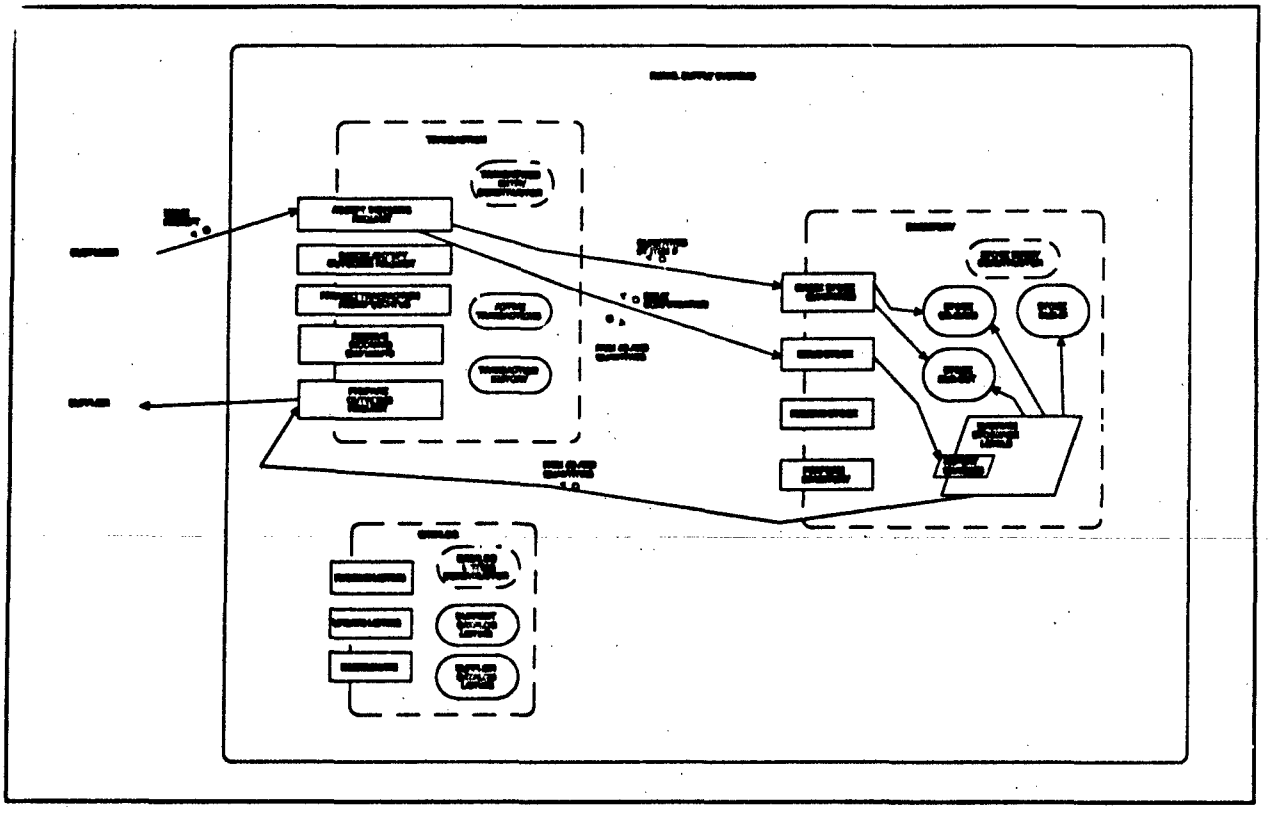


Figure 6. Supply Object Decomposition

3.0 SYSTEM INSTANTIATION

In order to implement a system with domain analysis and design products as inputs, the specific characteristics of a system are "overlaid" or instantiated (informally) to the generic templates. For generic requirements, system-specifics are gathered from business process improvement products (IDEF0 and 1X models), interviews with functional proponents, and user/site input. The system instantiation takes the form of insertions/modifications of system-specific information into the requirements template. For the generic architectures, any system-specific design features are added to (or updated to) the generic design to satisfy implementation constraints and system-specific requirements. The domain model and generic architecture products will contain information about reuse guidance, rationale, alternatives considered and discarded, and lessons learned from the systems' development and domain analysis efforts.

4.0 CONCLUSION

Domain analysis and design allow identification of software components early in the development life-cycle, including the planning stages. Orientation to a design and to the common requirements of a domain allows the components to be larger, more complex, and more highly integrated than traditional reusable components. These features can result in higher levels of software reuse in the applications within the domain of the architecture. These same features may make a generic model and architecture less usable outside of the application domain for which they were created.

In order to realize the savings required to meet budgetary cutbacks, a systematic way of identifying large-scale software reuse is necessary. Developing and using generic requirements and designs will make the difference in moving a system out of the initial planning stages into development and fielding.

AUTHORS

William (Will) Stewart is a member of the Reuse Engineering technical staff at SofTech. He has 10 years experience in software engineering, including work on Space Station Freedom and the Army WWMCCS Information System (AWIS). He has received degrees in both Computer Science and Engineering from Old Dominion University. His interests include methods to support domain engineering, knowledge engineering, and robotics. Internet: wstewart@softech.com

William (Bill) Vitaletti is a member of the Reuse Engineering technical staff at SofTech. He holds a M.S. in Computer Science and a B.S. in Business Economics from the State University of New York at Binghamton. His current interests focus on domain analysis modelling and automated tools development for use in the domain analysis arena. Internet: wvitaletti@softech.com

U.S. Mail:

SofTech, Inc.
1600 North Beauregard St.
Alexandria, VA 22311

REFERENCES

1 Vitaletti, William G., and Chhut, Ravinn, *Domain Analysis Guidelines (Draft)*, for the DoD Software Reuse Program (DISA/CIM), pp 1-1, 2-1, SofTech, Inc., 8 May 1992.

2 Chubin, Sherrie, DISA/CIM/XRE, *Domain Analysis Workshop Proceeding*, for the DoD Software Reuse Initiative, pp WG1-4, 21-22 September 1992.

3 Tracz, Will, IBM Federal Sector Division, "Domain Analysis Working Group Report - First International Workshop on Software Reusability", for ACM SIGSOFT, *Software Engineering Notes*, vol 17, no3, pp27-33, July 1993.

4 Vitaletti, William G., and Guerrieri, Ernesto, SofTech, Inc., "Domain Analysis within the ISEC RAPID Center", *Eighth National Conference on Ada Technology*, pp 1-5, Atlanta, GA, 5-8 March 90.

5 *"The Army Strategic Software Reuse Plan"*, office of the Director of Information Systems for Command, Control, Communications and Computers (ODISC4), 31 August 1992.

6 Bailin, S., "An Object-Oriented Requirements Specification Method", *Communications of the ACM*, pp. 608-623, May 1989.

7 Braun, C., "Domain Specific Software Architectures - Command and Control", *Proceedings of the WISR '91 4th Annual Workshop on Software Reuse*, 1991.

8 Buhr, R., *Practical Visual Techniques in System Design with Applications to Ada*, Prentice Hall, 1990.

9 Coad, P., Yourdon, E., *Object Oriented Analysis*, Yourdon Press, 1991.

10 Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A., *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report CMU/SEI-90-TR-21, ESD-90-TR-222. Software Engineering Institute, Carnegie Mellon University, November 1990.

11 Prieto-Diaz, R., "Domain Analysis for Reusability", *Proceedings of the Eleventh Annual International Computer Software and Applications Conference* (COMPSAC 87), pp. 23-29, Tokyo, Japan, October 7-9, 1987.

12 Quanrud, R., *Generic Architecture Study*, Technical Report 3451-4-14/2, prepared for U.S. Army CECOM, SofTech, Inc., January 1988.

13 McDonnell Douglas Missle Systems, *Developing and using Ada Parts in Real-Time Embedded Applications*, CAMP-3, prepared for USAF Armament Laboratory, April 1990.

14 Gilroy, K., Comer, E., Grau, J., Merlet, P., *Impact of Domain Analysis on Reuse Methods*, prepared for U.S. Army CECOM, Software Productivity Solutions, Inc., November 1989.

15 Bailin, S., "Towards a Case-Based Software Engineering Environment", *Proceedings of the WISR '92 5th Annual Workshop on Software Reuse*, Palo Alto, California, October 26-29, 1992.

16 *PEO-STAMIS Domain Definition Report*, prepared for the Army Reuse Center, Document No. 1213-65-210/3, SofTech, Inc., September 1992.

17 DISA CIM Technical Reference Model, Draft, 1992.

18 Vitaletti, William G., Stewart, William R., and Chhut, Ravinn, *Domain Analysis Guidelines (Version 1)*, for DoD Software Reuse Program (DISA/CIM), SofTech, Inc., December 1992.

19 *Design/Coding Guidelines for Reusable Ada Software*, prepared for DISA/CIM, Document No. 1222-01-210/9, SofTech, Inc., November 1991.

20 *ISEC Portability Guidelines*, prepared for the U.S. Army Information Systems Engineering Command, SofTech, Inc., December 1985.

A PRACTICAL GUIDE FOR ADA REUSE

Robert Haddon Terry
Margaretha W. Price

MountainNet, Inc.
Morgantown, West Virginia

Summary

Although reuse is accepted as a means of improving software quality and productivity, only a limited number of organizations are taking its acceptance seriously. Our paper discusses the processes involved in reuse implementation. It relies upon the recorded achievements and lessons learned from previous and current projects as the basis for its recommendations. This practical guide includes a discussion of reuse issues, actions, and benefits. With our proposed approach, an organization can gain confidence through a low-cost, modest reuse program, while developing valuable reuse expertise. Findings are presented in terms of products, which include information for initiating and managing a reuse effort.

Reuse in the Life Cycle

Several experts recommend the implementation of reuse throughout all phases of the software development life cycle. Dr. Kyo C. Kang of the Software Engineering Institute proposes refining the life cycle model to include reuse activities at each phase.[1] Dr. Charles McKay, a NASA scientist, emphasizes the importance of developing, importing, classifying and leveraging reusable components throughout the Space Station Program life cycle.[2]

Even though it may be apparent that reuse activities should be included in all phases of the development life cycle, economic constraints and managerial attitudes have limited its implementation. We have, therefore, elected to limit our scope of concentration to the code development phase.

In the coding phase, visible benefits are more readily identified. Reuse at this phase can easily be defined by the ratio between DLOC (Developed Lines of Code) and RLOC (Reused Lines of Code). The benefits of reuse are not so immediately evident or measurable during the other life cycle phases. For instance, during the design and testing phases, due to the artistic nature of the activities involved, it is difficult to calculate time spent or saved.

Before we outline our recommended approaches, let us first report on our investigation of the current issues, experiences encountered, and lessons learned by other reuse efforts.

Investigation Results

Questions Encountered

Selecting a practical, achievable reuse approach is facilitated by a clear understanding of the issues relevant to adoption and implementation. These issues and questions include:

- Development Methodology - During which life cycle phase, e.g., coding, design, testing, and/or documentation, should reuse approaches be applied to achieve maximum efficiency?
- Analysis Techniques - Which of the analysis techniques produce intuitively clear and achievable paradigms of a practical reuse approach?
- Assessment Approaches - Which assessment approaches best identify those reusable artifacts most relevant to a given developer's needs?
- Multiple Instances - How should multiple instances of reusable artifacts be constructed?
- Integration - How best are the aforementioned activities integrated into the other life cycle phases?
- Presentation - Which presentation methodologies, e.g., modeling and diagramming techniques, best suit a given developer's needs?
- Testing - Which testing methods most adequately ensure the integrity and viability of the approach selected?
- Management - How are policies regarding research, quality control, and information dissemination defined and implemented?

Practical Approaches

Certain basic steps and practices have been effective in addressing and overcoming reuse implementation challenges. The following briefly describes proven, fundamental measures which support sound approaches.

- Investigate the systematic and limited use of existing reuse successes. In particular, apply theories, which are just sets of axioms, to activities, thereby utilizing the most current actual advances.
- Set a clear distinction between small-, medium-, and large-scale reuse opportunities.
- Provide the reuse effort with a comprehensive variety of software development information.
- Effectively and definitively rely upon reuse successes to maximize the reuse effort.
- Provide support for different development methodologies at different stages within each model.

Recommendation: Reuse Expert (RE)

The above suggestions are certainly warranted and needed, although they lack a means for implementation. One solution is to create a new position called a Reuse Expert (RE).

Suggested qualifications and responsibilities which the RE should attain over time:

- Be a full-time member of the development team, whose sole responsibility is to the reuse effort itself.
- Be a senior staff member who is held in high regard by the development team and displays a positive attitude towards reuse.
- Participate in the analysis, design, and coding phases of the development process.
- Attend all team reviews and walk-through demonstrations to facilitate constructive communication and to avoid duplication of effort.
- Research, analyze, and report on reuse-related resources.
- Obtain training in areas relevant to his experiences.
- Maintain the common and shared in-house software repositories, monitoring access and growth.
- Avail himself of and exploit all opportunities in training the development staff in relevant reuse materials.
- Communicate reuse development needs to management and gain expertise great enough to warrant shifts in the development paradigm.
- Expose his findings to the reuse community by publishing, demonstrating, or attending current forums.

Information Gathering

We began by performing a thorough investigation of possible candidates. Quality controls included verification and validation of sufficient source documentation. We then attempted to identify and locate supporting material for the list of chosen possibilities. The resulting materials were analyzed and initial candidates identified using the defined quality measures.

We then created and issued an informal survey. The survey included questions regarding the current level of reuse, the number and position of persons involved, and the nature and instances of reliance upon repositories versus the availability of COTS (Commercial-off-the-Shelf) components.

We originally intended to target the survey to six or more high-level managers or, at the very least, those individuals most likely responsible for a reuse endeavor. After a series of frustrating contacts with voice mail systems, answering machines, secretaries, and beleaguered colleagues, we concluded that access to this target audience was not only exhausting but difficult to obtain. However, those surveys which were completed and returned were scrutinized to be sure the answers provided could be verified. We then analyzed the responses and selected three Ada software development projects which not only represented the variety of available development approaches but also displayed a penchant for reuse. Again, quality guidelines were established and included such issues as size, direct contact with members integral to the effort, and the receptiveness of these individuals to incendiary suggestions and comments.

Data Presentation and Analysis

In order to preserve audit objectivity, we elected to present the information anonymously. Although we have been granted permission to make full disclosure of our findings, particularly since most are currently and readily available, we feel it serves no purpose to attach our suggestions to any particular endeavor. We are attempting to offer solutions which could be used by any effort. Indeed, it is our hope that these scenarios result in a better understanding of reuse, its properties, and its acceptance.

Project A

General profile: A multiuser information system encompassing an inventory control system with command decision modules.

Demographic data: The system consists of 300,000 LOC (Lines of Code) and took approximately two years to develop. Testing was slated for January 1992. A total of 25 assigned software developers have been involved on a full time basis. There were also more than five support personnel and some outside contracting. The developers had different roles including three system administrators, four communication specialists, two database administrators, one configuration manager, and more than five teams of 2 to 3 application developers.

Reuse achievements:

- New development efforts have adopted their methodology, standards and software architectures.
- A scarcity of off-the-shelf software required the development of an Ada configuration management system which will also be exported to other systems.
- The methodology has been documented in a much needed "Developer's Guide". The guide shows potential as an organizational standard.
- Approximately 6,000 lines of locally written, reusable code (12% of the total lines) equals 30% of the system based on the number of times they have been reused.
- A page formatting component was reused from the ASR (Ada Software Repository).
- GRACE Components were purchased and used, albeit not extensively.
- Preliminary discussions have begun about their own in-house repository.

Reuse lessons learned:

- Lack of tools. Very few commercially available tools had been developed which appropriately fit their needs.
- Unanticipated tasks. Searches for reusable artifacts had to be conducted, prices negotiated, and testing of the procured products had to be accomplished.
- Lack of resource knowledge. Greater emphasis should have been placed upon searching Ada repositories for specific software requirements and developing packages.
- Lack of component support. A support group for the purchased reused components was created and later disbanded. The product was never made fully available.
- Program reviews were attempted to increase the amount of information sharing, but they were discontinued because of time and coordination constraints.
- Unanticipated ramifications. By judicious integration of the database, control files, and reusable packages, varying field requirements and development modifications were affected by minor system tuning.

Recommendations:

- The project should have retained one RE.
- The RE should have seen to the organization and productivity of the program reviews.
- The RE should have coordinated and managed reuse component acquisition and purchase.
- The RE should have constructed an in-house repository beyond the initial and immediate needs of the current development project.
- The RE should have been responsible for repository searches and retrievals as encountered.

Project B

General profile: A financial system which is an application redesign that services an accounting and finance center.

Demographic data: The system consists of 2.75 million LOC with a total development time of 4.5 years. Of this total, approximately two years were spent coding. The system was delivered to the customer after development and qualification testing was completed. A total of 90 people, including 52 coders, were directly involved at the peak period. Responsibilities were divided among management, database, senior analyst, system support, advanced technology, application, design, and application generator teams.

Reuse achievements:

- Portions of the design were isolated into packages. This permitted a physical decomposition of code rather than decomposition along functional lines.
- Timing issues were encapsulated in a single mainline generic package for reasons of consistency. Over time, advantages were found by having several mainline generic packages. A benefit of this approach was that every functional subsystem had the same "look and feel".
- A follow-on inventory project, about one-third the size, used the tools created and reduced its cost by one-half.
- Each of the more than 2,500 data items was defined in a database. Centralized management of data item typing enforced an uncommon level of consistency throughout the code.
- All interactive programs had, by definition, exactly one screen. A screen painting tool was used to generate the code necessary for the screen and its system mapping support.
- Specifications were used as input into a generator which generated bodies. A full 76% of the 2.4 million lines of code were generated.

- Code sharing was supported by creating, distributing, and using common and shared libraries. Walk-through provided opportunities to share the knowledge of libraries' contents.
- Common code (e.g., a menu package) and shared code (e.g., date operators) were maintained and made available.

Reuse lessons learned:

- Lack of communication. Communication between library users was difficult to maintain and facilitate. It was partially alleviated by an application leader who assumed monitoring responsibilities.
- Limited use of resources. Use of the Booch components and ASR was minimal since the developers feared their introduction would result in configuration management problems.
- Lack of coordination. The powerful senior design team did not communicate the properties of reuse effectively.
- Unanticipated ramifications. Later in the development, project reuse efforts were consistently impeded by database difficulties.
- Lack of integration. Walk-throughs became internal team meetings while information sharing was limited to segregated groups. There was very little reuse between groups at the application level.
- Lack of in-house repository. The effort lacked an in-house repository, which could have also served as a storage unit for the specifications used with the generator.

Recommendations:

- The project should have retained two REs.
- The REs should have made intra-team communication a priority and should have been present at each walk-through.
- The REs should have taken full advantage of the reuse opportunities provided by the Booch components and the ASR.
- The REs should have maintained, monitored, and disseminated the common and shared library contents.
- The REs should have been included in the decision making sessions held by the senior design team.
- The REs should have planned for later reuse efforts by automating data changes with tools that could handle increased populations.

Project C

General profile. An advanced automated system for aviation. The system is presently broken into 16 logical groups, which are comprised of about 45 large, distributed processes.

Demographic data: The final product will require an estimated 2.5 million Ada LOC, plus some JOVIAL, C, and Assembler software mixed with COTS. The system is currently under development and has a development life cycle estimated to last 12 years with a maintenance "phase" greater than 30 years. There are over 1,000 employees from multiple companies working on the project. The surveyed company's involvement includes 230 people who work in approximately 45 groups, one of which is a reuse working group (RWG) that has 13 full-time and 10 part-time participants.

Reuse achievements:

- An ad hoc working group was set up, as a task force of interested volunteers, to determine how the project could take advantage of reuse technology.
- The working group evolved into an official organization which provides a project-wide coordination of the reuse effort.
- The RWG defined a reuse process and established responsibilities for each of its members, which are documented in the project's development standards.
- The RWG defined roles and responsibilities for software authors, reusers, and reuse advocates. These include items which assure the quality of the reuse products, processes, components, and people.
- Using a literary warrant technique on the in-house components, the RWG performed an abbreviated domain analysis. Initially, 105 different classes were identified that could be potentially (re)used by two or more development groups within the project.
- The RWG tracked the development, use, and changes of the reusable components. They advertised the components and their "review" schedules.
- Each development group is "surcharged" for reuse by taking out the cost of the Reuse Group from each of the development groups' budgets. It is hoped they would take advantage of what they are paying for.
- The sharing of software (up to 9 times in one case) forced the sharing of information between different development groups, which increased the understanding of the problem domain and raised each participants' levels of expertise.
- The RWG searched various forums and repositories (e.g., in-house repositories, AdaNET, COSMIC, GRACE, and STARS).
- The RWG recommended "rewards" and other incentives for participation in component reuse.

- The RWG members attended all project reviews and sought out opportunities for reuse
- The RWG trained people about the reuse activities at their reuse center, they also prepared and compiled surveys from users

Reuse lessons learned:

- The "traditional" software development process must be modified to incorporate the reuse effort
- It was discovered that more granularity in tracking data was needed. Originally, they only tracked reuse at the "component" level
- Document standards have to be tailored for reuse, since they typically require inappropriate information and formats
- There needs to be more automation of the process
- Reuse must be considered early in the life cycle, at the specification, or even the proposal stage
- Areas where potential reusable components are identified would need extra funding to develop the components

Recommendations:

- The RWG should continue making progressive steps in the reuse efforts
- The RWG should sub divide their efforts in order to reflect their concerns with the proposal, design, development, and maintenance of the product
- The mission of the RWG should further be a comprehensive reuse effort, and not be associated with any particular attempt
- Particular roles within each sub group need to be identified and emphasized (e.g., RE, Domain Analysis, Domain Expert, etc.)
- Their effort could include (maybe should) at least four RE's

Product 1: How to Start A Reuse Program

Misconceptions regarding reuse have hampered acceptance. Far too many developers adhere to the myth that reuse is a costly and generally risky business. Projects are quite often overly ambitious with ill defined goals and inflated expectations. However, if the effort is modestly undertaken and based upon a focused, concentrated, and clearly stated mission, the overall costs will be negligible. Balance those costs against the practical experiences learned, and the arguments against reuse efforts are less compelling.

Selecting and supporting a qualified, enthusiastic, and responsible team leader are essential to any effort. For our purposes, this person is designated the RE. Removing the expert from an existing team and positioning him in this new lead role can be viewed as an initial investment. The number of reuse experts necessary must be measured against the staff and project size. The number of RE's may also grow in response to project growth and maturity. For most modest projects, one full-time, committed expert is sufficient. This commitment must be total, since the responsibilities involved in reuse demand constant evaluation, intervention, and control. Further qualifications are detailed in following sections.

Once appointed, the reuse expert will then be required to evaluate in-house packages and components. We have adopted the term "objects" to represent packages and/or components. Companies with extensive object collections should make judicious selections finding objects that match the effort's goal and mission. Smaller firms who have not produced vast amounts of code can choose to experiment with all the objects collected.

Ideally, a comprehensive domain analysis would be performed at this point. Our experience indicates that an "abbreviated domain analysis", performed by the RE, is more practical and useful.

This entails using a form of the literary warrant technique employed by library scientists. In this technique, existing objects are grouped according to their commonalities.

A brief example

Firm XYZ has the following seven objects:

- simple editor
- text formatter package
- random number generator
- array manipulation
- spell checker
- binary tree functions
- matrix functions

Performing an abbreviated analysis could result in, but is certainly not limited to, the following grouping:

Group, Text Manipulation
Simple editor
Text formatter package
Spell checker

Group, Mathematical Routines
Random number generator
Matrix functions

Group: Abstract Data Types
Array manipulation
Binary tree functions

Obviously, there are many methods of analyzing and classifying objects using a variety of nomenclatures.

Computer Science Corporation's Reuse Working Group, for example, did something they called a "mini domain analysis" which was successful in identifying common classes and facilitating communication.[1]

With the existing objects grouped, the reuse expert can then discuss their relevance with developers who will be encouraged to rate the necessity of each object to their current project. Developers will also be asked to produce a list of objects for possible acquisition or development. This process is vital for building confidence in the overall reuse effort.

Developers' feedback will provide the reuse expert with two lists of objects:
1. in-house objects deemed necessary to the project at hand
2. "in demand" objects which must be located, and retrieved, purchased or developed

If there exist in-house objects which developers think they need, they can just make a copy and reuse it. Otherwise, the developer meets with the RE to decide whether a form of the object is on the "in demand" list and whether the demand is great enough to warrant developing it as a reusable object at this time. If it is appropriate to develop it as a reusable object, the RE will organize and coordinate the effort with the other potential users of the object. The coordination would include identifying the common requirements and deciding on which developer could handle the extra work load.

Management needs to encourage the development of reusable objects by allotting more time for the effort. Developers need to share the responsibility of creating reusable objects.

Product 2:
RE Responsibilities and Activities

Qualifications of the candidate:

Candidates need to be experienced in the development process. Ideally, they should also be students of the reuse effort. They must possess the abilities to offer and promote procedural shifts in the organization's developmental paradigm. Finally, they must be prepared to manage and enlarge upon the reuse effort.

Three recommended levels of expertise:

Beginner

Focus: Research and analyze what has been done, focus attention towards what is needed. Complete the following recommended activities until satisfied of success.

Activities:

- Gather and maintain information about the in-house software domain.
- Gather and analyze information concerning the reuse paradigm shifts.
- Contact and interact with any/all potential component sources for your domain of interest (e.g., repositories or commercial components).
- Perform an abbreviated domain analysis of in-house artifacts.
- Locate and enroll in one or more reuse workshops/tutorials.
- Gather and analyze information about related subject areas (e.g., Domain Analysis, Designing for Reuse, Library Sciences, etc.) as they are encountered and become relevant to your effort.
- Attend and participate in any/all program reviews, design reviews, etc.
- Reproduce and distribute all relevant products (e.g., catalog of in-house components, report of external components, etc.).

Intermediate

Focus: Continue efforts in researching, analyzing, documenting, and reporting. Begin introducing reuse as an alternative to development, focus on easily understood and worthwhile attempts.

Activities:

- Prepare for workshops/tutorials, read relevant publications of the speakers or the subject. Communicate any questions and special interests with the speakers (allowing them time to address the questions in the session).
- Attend workshops/tutorials, obtain and briefly review all the provided material. Ask questions as they arise, offer suggestions when solicited. Review and prepare for each session (e.g., during breaks or overnight stays).

- Create a summary report of lessons learned from the workshops/tutorials. Include ideas which might be implemented in your company.
- Continue attending and participating (more actively) in in-house program reviews. Seek out and select one or more opportunities for reuse.
- Keep accurate records on the selected reuse opportunities. Track hours, LOC, and other data which can be used to evaluate the reuse effort. Record all data in a spreadsheet package.
- Propose the purchases of commercial components, if their contents fulfill your domain needs. Include the components in the in-house library and distribute information about them to potential users.
- Reiterate the abbreviated domain analysis, to provide finer granularity and to identify obvious omissions.
- Create a taxonomy (listing) and an architecture (model) of the in-house library.

Expert

Focus: Continue with the carry over activities of the prior levels. Propose instances where reuse should occur. Use obtained data to statistically demonstrate the value of reuse.

Activities:

Conduct reuse training when appropriate and needed. The training should be short and directed towards the audience's abilities.

- Provide management with a reuse activity report. Include statistics that demonstrate the value of the effort. Include paradigm shifts which would improve the process.
- Create a reuse working group (possibly from volunteers) who have the qualifications needed to improve upon the reuse effort.
- Perform an abbreviated domain analysis of the external resources previously identified. Prepare a taxonomy (include price information) and an architecture of the ones which are part of your in-house domain.
- Propose and solicit places where reuse is needed from the working group members. Prepare a report which demonstrates the potential value of your suggestions. Deliver the report to those in charge of making such decisions.
- Attempt to publish your experiences and findings.

Product 3: Recommendation and Consideration For Management

Our paper thus far has introduced a basic reuse program involving minimal start-up costs. This basic introduction can and should be expanded upon to achieve maximum benefits. This expansion will be made possible by an increased level of expertise and effort maturity. Moving the effort forward is a natural progression for both the RE and the project itself. Figure 1 illustrates this point by combining the steps outlined in the Section , "How to Start A Reuse Program", and the following Section which tracks the RE's expanding level of expertise. The figure also provides cost and time estimates for each iteration necessary to achieving the ultimate goal.

The time consumed by increased reuse activities depends upon a variety of factors which impact project decisions. Renown reuse and domain experts such as Ruben Prieto-Diaz support the idea of progressive reuse, e.g., reuse achieved through a series of stages based upon underlying

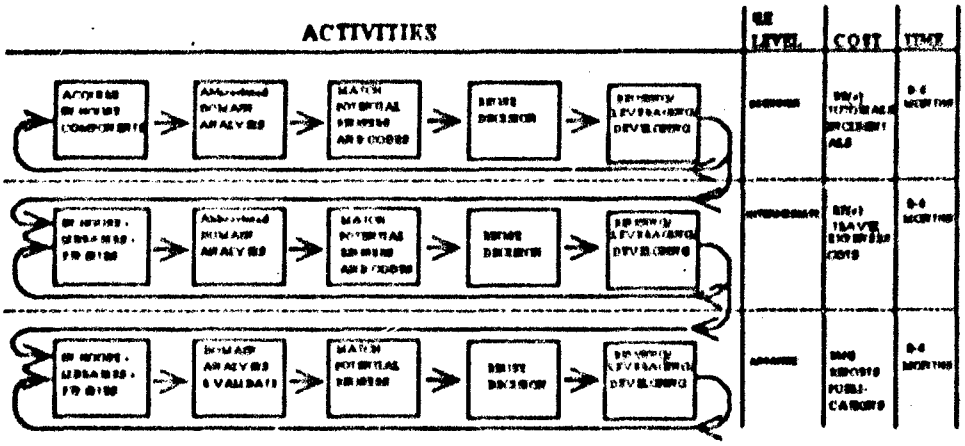


FIGURE 1 REUSE PROGRAM STAGES

success at each level.[4] We concur and support this contention graphically in Figure 1. Note that allowances are made for both progressive and horizontal iterations. Horizontal iterations are performed when unsatisfactory results occur at any given level.

Figure 1 also shows that cost allowances increase as the project matures. This is typical of reuse efforts and, as noted by Prieto-Diaz, most projects will experience fluctuating costs before stabilizing. Figure 2 represents typical cost behavior.[4]

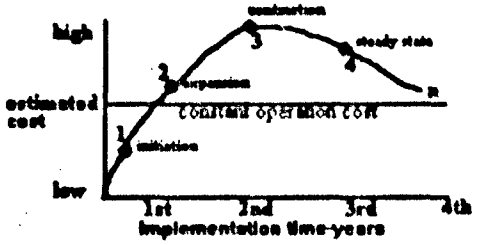


FIGURE 2 - Expected Cost Behavior for a Typical Reuse Program

The most important factor influencing the success or failure of a reuse effort is undoubtedly the RE, his commitment and expertise. Management must enforce, if necessary, the total involvement of the RE in all aspects of the software development life cycle. The number of REs involved in the project depends upon the nature and level of the project itself. Any reuse effort will be shortchanged if management underplays the importance of this role.

As reuse is integrated further into the development process, the number and involvement of REs increase accordingly. The RE's role should expand in all other phases of the software development life cycle.

Management should support the reuse effort by rewarding the development team with creative incentives. Developers who successfully reuse code or create reusable code should receive management recognition or tangible rewards. Management should make available articles, papers, and any current information on the subjects of reuse, and should encourage their utilization. One-hour, weekly status meetings should be scheduled to facilitate the information sharing on the reuse effort and its impacts on current projects.

Product 4:
Available Resources

The following is a representative sampling of available resources. It is meant to serve as a good starting point for beginners and as template for the data structure.

File: Commercial Components

Record Name: Booch Components
Vendor: Rational
2171 South Parfet Court
Lakewood, CO
Phone: (303) 986-2405
Contact: Grady Booch
Abstract: Consists of several dozen domain independent data structures and tools, each with multiple implementations so that a client can select the representation that provides the most suitable time and space characteristics. Although written in Ada, Versions are planned for C# and SmallTalk. Over 300 sites in the US, Europe, and Pacific Rim.
Vendor Profile: Rational is recognized as the world's leading supplier of Ada products and services. Grady Booch is Rational's director of object-oriented product development as well as the developer of the Booch Methodology and Booch Components.

Record Name: The GRACE (Generic Reusable Ada Components for Engineering)
Vendor: EVB Software Engineering, Inc.
5303 Spectrum Drive
Frederick, MD 21701
Phone: (800) 695-1815
Fax: (301) 695-7734
Contact: Jennifer Jaynes Lott
Abstract: Consists of 275 distinct components organized into 23 families of abstractions which total more than 520,000 lines of Ada Code. GRACE components and the 18,000 pages of documentation are provided on magnetic media.
Vendor Profile: EVB provides consulting services to assist management and technical teams in the absorption and utilization of software engineering technology. Also provides an extensive Ada Software engineering training curriculum, products that enable and enhance Ada Software development.

File: Information Services

Record Name: Ada IC (Ada Information Clearinghouse)
Address: 4600 Forbes Boulevard
Lanham, MD 20706-4320
Phone: (800) AdaIC-11
Fax: (703) 685-7019
Contact: Susan Carlson
Internet: ajpo.sei.cmu.edu
Abstract: Sponsored by the Ada Joint Program Office (AJPO), the AdaIC Bulletin Board makes information on the Ada language available to the public. Sample flyer topics include: "Ada Source Code, Reusable Components and Software Repositories", "Costing, Sizing and Productivity Tools", and "Ada Training Videotapes Available through National Audiovisual Center." The read.me file provides a complete listing of files available for downloading or transferring.

Record Name: archie
Internet: quiche.cs.mcgill.ca
Login: archie
Password: (name needed)
Contact: The "Archie Group" of McGill University
Bill Heelan (wheelan@cs.mcgill.ca)
Peter Deutsch (peter@cc.mcgill.ca)
Alan Emtage (bajan@cs.mcgill.ca)
Abstract: archie is an interface for use with anonymous FTP resources. It is a system which allows you to rapidly locate the various public domain programs stored on the hundreds of sites across the internet.

File: Repositories

Record Name: Army Reuse Center
Address: Army Reuse Center
USAISSDCW, Attention: ASQB-IWS-R, STOP H-4
Fort Belvoir, VA 22060-5456
Telephone: Client Services (703) 285-6272 or DSN 356-6272
Roy Lloyd (Newsletter Editor) (703) 285-9071 or DSN 356-9071
Abstract: formerly known as RAPID. Contains 1,401 Reusable Software Components (RSCs) totaling over 955,000 LOC. Current population strategy for FY93-FY95 will focus on identification, evaluation and certification of reusable components.

Record Name: AdaNet
Address: MountainNet, Inc.
2705 Cranberry Square
Morgantown, WV 26505
Telephone: Help Desk (800) 4 ·· ·8
Internet: adanet.wvnet.edu.
SprinNet: 304130
Abstract: AdaNet repository is a co onent of the Repository Board Software Enginee· · (BSE) program. The program provides a possible doma ·euse library with software from the ASR, Jet Propuls· n Lab (NASA/ JPL), DOD/STARS, and Education Institutions.

Conclusions

Our study has communicated, managed and qualified the reuse effort. In near retrospect, it has succeeded as an initial analysis of the reuse domain. We contend that it is worthwhile reading for individuals, groups, and organizations involved in the reuse effort. We are comfortable in the thought, that our products might provide usable information in future attempts.

Acknowledgments

The assistance of the many contributors to this study is greatly appreciated. The authors wish to thank Linda K. Braun, Debra Burns, Joyce Coombs, Karen Fleming, Kathy Uphold, LouAnn Welton and Debbie Zewe for their valuable assistance in collecting data and editing this paper. We also want to acknowledge the AdaNET project and MountainNet Inc. for their support and encouragement.

References

[1] Kyo C. Kang, "A Reuse-Based Software Development Methodology," Proceedings of the Workshop on Software Reusability and Maintainability pp.194-196, October 1987.

[2] Charles McKay, "Final Report on "A Study to Identify Tools Needed to Extend the Minimal Toolset of the Ada Programming Support Environment (MAPSE) to Support the Life Cycle of Large, Complex, Non-Stop, Distributed Systems Such As the Space Station Program," 1990.

[3] Gregory M. Bowen, "An Organized, Devoted, Project-Wide Reuse Effort," Ada Letters, pp. 43-52, January/ February 1992.

[4] Ruben Prieto-Diaz, "Making Software Reuse Work: An Implementation Model," ACM SIGSOFT Software Engineering Notes, Vol. 16 no. 3, pp. 61-68, July 1991.

Biographies

Margaretha W. Price; MountainNet, Inc., 2705 Cranberry Square, Morgantown, WV, 26505. She is a Software Engineer/Librarian in MountainNet, Inc., working on the NASA/AdaNET project. She received BS (1990) and MS (1992) degrees in Computer Science from West Virginia University, Morgantown, West Virginia.

Robert Haddon Terry; MountainNet, Inc., 2705 Cranberry Square, Morgantown, WV, 26505. He is a Software Engineer/Librarian in MountainNet, Inc., currently working on the NASA/AdaNET project. He received a BA (1984) degree in Mathematics from West Liberty State College, West Liberty, West Virginia, and a MS (1988) degree in Computer Science, from West Virginia University, Morgantown, West Virginia.

Engineering Environments & Emerging Standards

Moderator: Frank Belz, TRW

Panelists:

Ada9X

Moderator: Chris Anderson, US Air Force

Panelists:

Reuse Education

Moderator: Rose Armstrong, DSD Laboratories

Panelists: Dr. Dave Eichmann, University of Houston, Clear Lake
Dr. Charles McKay, University of Houston, Clear Lake
Bob Saisi, DSD Laboratories
Linda Saus, EWA
Dr. Francis Van Scoy

AdaSAGE

Moderator: Joan McGarity, COMNAVCOMTECCOM

Panelists: Howard Stewart, EG & G Idaho Inc.
David Cuneo, Naval Computer & Telecommunications
Capt. Jerry Depasquale, US Marine Corps
Am Hollom, Standard Systems Center
John Taylor, Software Development Center, US Army

Government Training for Ada & Software Engineering

Moderator: Capt. David A. Cook, US Air Force Academy

Panellsts:

Software Reuse: The Next Steps

Moderator: James Hess, HQ, Department of the Army

Panelists: Phillip L. Koltun, Harris Corporation
Robert Lencewicz, HQ, Electronics Systems Center
Donald Reifer, RCI
Roger Williams, Software Productivity Consortium

USING ADA FOR A TEAM BASED SOFTWARE ENGINEERING APPROACH TO CS1

AKHTAR LODGHER & JAMES HOOPER

Department of Computer Science and Software Development
Marshall University
Huntington, WV - 25755
Phone: (304)-696-2695 Fax: (304)-696-4646
Internet: CIS005@marshall.wvnet.edu

In the past year, the Computer Science department at Marshall University has revised the Bachelor's degree program, and given a very strong emphasis to software engineering throughout the entire curriculum.[1] The department decided to use Ada as the standard programming language for the first few courses. In later courses, exposure to other languages such as C and C++ is also given. The program has two capstone courses, taken in the last two semesters, where a major team project is designed and implemented. Hence the need for emphasizing software engineering principles, as well as getting students used to programming in teams from the very first computer science course was strongly felt. In this paper, the author presents the syllabus and a method of executing the syllabus of the CS1 course satisfying the above needs. Software engineering principles are introduced early on, and after an initial boot-strapping period, the programming projects are done in teams. The Ada programming language is used.

Introduction

CS1 is taught as a 4 semester-hour course in a 16 week semester. The students attend three hours of lecture a week and two hours of closed lab. Concepts introduced in the class are reinforced in a closed lab setting. An open lab is also available for students to complete their lab exercises and programming projects. The Ada compiler on a VAX/VMS system is used. Students are allowed to use PC based compiler environments for program development. However all work is graded only on the VAX.

Syllabus

The objective of this course is to develop problem analysis and algorithm development skills. Topics covered in this course include introduction to the entire life cycle of software development, introduction to the use of modular design in the problem solving process, procedural abstraction, decision structures, iteration structures, basic data types, array and record structures, abstract data types, use of generic code, and introduction to dynamic structures. Problems that enhance the characteristics of each concept/structure are used. The problem solving process is emphasized over language implementation. An example of this principle for illustrating the looping process is: "Let us study this problem (which requires a loop construct) and develop an algorithm for its solution" rather than "These are the looping constructs available in this language. Let us see the kinds of problems that can be solved using these constructs".

The solution process of these problems is studied strictly from a software engineering perspective -- conducting a requirements specification and analysis, performing a modular top-down design, development of module specifications, adherence of code to design. From the very first class, the students are told to perceive themselves as software engineers

and designers, not programmers.

A team approach is used for programming assignments - two students per team. One person of the team does the design and the other person develops the code based on the design. For the next assignment, the roles are switched. This approach forces the designer to conduct a proper analysis and design. The "coder" has to follow the design, making only necessary changes, if required.

Approach

The "hands on" approach

The amount of material covered in the class is quite large. To ensure that enough exposure is given to each topic, a "hands on" approach to instruction is used. Programs which exhibit the characteristics of a particular concept or structure are made available to the students. These programs are displayed, explained, and executed in the classroom, on a computer using the overhead video projector. Unlike the traditional "chalk-and-talk" approach, this approach not only shows the syntax of the structure, but also shows how the structure is used in the context of a larger solution process. Minor variations and nuances of the structure are also explained.

Another important advantage of this approach is to show the possible incorrect ways of using the structure. When a student starts using a new structure, the chances of him or her using it incorrectly are high. By using incorrectly formed structures (both syntacticaly and semantically incorrect), the error messages generated are shown. The mechanism of using the error messages to trace the error in the structure can be demonstrated.

Class notes in electronic form, as well as all classroom demonstration programs are made available to the students (on a mainframe) before the class. The students are encouraged to bring a printout of the notes and the programs to the classroom. This allows them to spend time listening and participating in the classroom discussion and not be bogged down by the task of taking notes.

Other advantages of the hands on approach inc increase in student participation (answering "wha questions), increase in understandability a id incr in programming confidence. However, this appr places a tremendous burden on the instructor. development and preparation of pedagog examples takes a lot of time. Instruction mate associated with text books are not available electronic form. Such material must either scanned or typed in and fine-tuned depending u the audience.

Course contents and weekly topics

Table 1 shows the classroom topics, the assignr and lab topics on a weekly basis. It is assumed the student has little or no knowledge of operating system. However, it has been found f past experience that students who have had introductory course on computers in high schoo more patient and quicker in learning the operating system.

The first two weeks introduce the entire system cycle of software development. A top-down anal and design methodology is discussed next. process of converting a problem statement requirements specifications, analysis and design simple problems is explained. Currently the anal is done using data flow diagrams (DFD's) and design using structure charts. A design man which explains this process in a step by step fash is made available. The issue of using object-orie design is under consideration.

The use of functions, procedures and package introduced early on, in the context of mod design. All the intricacies of procedures packages are not covered at this point. Only concept and their usage in simple contexts covered. The branching and looping constructs covered next.

Exception handling and the more detailed use functions and procedures are then explained. concept of abstract data types is introduced. ' array and record structures are covered n Examples of the use of array and records

Class Topics	Assignments Designs due beginning of week Programs due middle of week	Laboratory Laboratory work is due end of week
WEEK 1: - Introduction to problem solving - Software Engg Life cycle		LAB 1: - Introduction to the operating system - File structure, network characteristics
WEEK 2: - Introduction to computational problem solving - Overview of high-level languages, syntax, semantics		LAB 2: - Operating system continued, Ada environment - Use of editor, batch files
WEEK 3: - Top down design, drawing DFD's, structure charts		LAB 3: - Simple Ada programs, DFD's, input, output
WEEK 4: - Introduction to imperative languages - Scalar data types, I/O formatting - Constants, variables, types	- Handout A1 (simple program)	LAB 4: - Run simple Ada programs, show execution of A1
WEEK 5: - Introduction to the use of functions, procedures and packages	- Design of A1 is due, program of A1 is due - Handout A2 (I/O formatting)	LAB 5: - Discuss A1, show execution of A2 - Building programs, debugging
WEEK 6: - Boolean expressions, concepts of branching - IF-THEN-ELSE statement, CASE statement	- Design of A2 is due, program of A2 is due - Handout A3 (Branching)	LAB 6: - Discuss A2, show execution of A3 - Use of IF-THEN-ELSE and CASE statements
WEEK 7: - Use of repetetion, LOOP - EXIT - FOR Loop, WHILE Loop, Nested Loops	- Design of A3 is due, program of A3 is due - Handout Team A4 (Procedures and loops)	LAB 7: - Discuss A3, show execution of A4 - Use of Loops
WEEK 8: - Enumerated types, attributes of scalar, float types - Parameters to procedures and functions - Error handling (exceptions)	- Discuss A4 - Handout Team A5 (Exceptions, packages)	LAB 8: - Discuss A4 - Attributes of scalar and float types - Procedures and functions

Table 1: Weekly schedule of classroom topics, assignments and laboratory exercises

Class Topics	Assignments	Laboratory
WEEK 9: - Visibility issues - Abstract data types, packages, file I/O	- Design of A4 is due, program of A4 is due	LAB 9: - Discuss A4, show execution of A5 - Exceptions, packages
WEEK 10: - Use of arrays, strings multi-dimensional arrays	- Design of A5 is due, program of A5 is due - Handout Team A6 (Arrays, records, file I/O)	LAB 10: - Discuss A5, show A6 - Arrays, multi-dimensional arrays
WEEK 11: - Use of records, hierarchical records - Variant records	- Discuss A6	LAB 11: - Discuss A6 - Records, hierarchical and variant records
WEEK 12: - More on packages, passing exceptions - Generics	- Design of A6 is due, program A6 is due - Handout Team A7 (Generics, exceptions)	LAB 12: - Discuss A6, show execution of A7 - Packages, generics
WEEK 13: - Generics continued - Intro to access types	- Discuss A7 - Handout Team A8 (Access types)	LAB 13: - Discuss A7 - Generics
WEEK 14: - Access types continued	- Design of A7 is due, program A7 is due	LAB 14: - Discuss A7, show execution of A8 - Use of dynamic variables (Access types)
WEEK 15: - Review	- Design of A8 is due, program A8 is due	LAB 15: - Discuss A8 - Access types continued

Table 1: ..continued

implement abstract data types are explained. It is at this point that a more detailed explanation of packages, passing exceptions, etc., are discussed.

The concept of code abstraction is explained using the generic structure on a sorting example. Finally, an introduction to dynamic data structures is given. The creation of dynamic variables and their use in creating linked lists and traversing linked lists is covered. It should be noted that the concept of recursion is not introduced in this course.

Assignments and laboratory exercises

A total of eight programming assignments are given. Of these, the first three are of an introductory nature and are done on an individual basis. The latter five assignments are done in teams. The first assignment, which is not given until the fourth week of classes, is of the nature of a "hello world" program. The second assignment involves some output formatting and the third assignment is based on the use of selection statements.

Each assignment requires the preparation of a design document. This document consists of: (a) the problem statement (b) requirements specifications (c) analysis - the data flow diagrams (d) design - structure chart showing the modules (e) module design specifications indicating the input, output and processing of each module. The design document is mandatory and must be submitted before starting the code. The code is based on the design, and the close relationship between the structure chart and the actual code is emphasized. The simple nature of the first three assignments helps in ironing out the details and links between design and code.

Beginning with the fourth assignment, the size and the complexity increases. At this point the class is divided in teams of size two. The members are chosen using a draw. One person is responsible for the design document and the other is responsible for the code. The roles for the next assignment are then switched. The following policy for grading team based assignments is set:

1. Essentially, each person gets the grade for the work done by him/her. Each assignment is worth 100 points.
2. If the design document is perfect, then the designer gets 100 points.
3. If the code follows the design and is perfect, the person in charge of code gets 100 points.
4. If the design is correct, and the code is incorrect, points are taken off from the coder.
5. If there are flaws in the design document, the designer loses points.
6. If there are flaws in the design, and the coder codes it following the design (resulting in badly designed code, though correct) the coder is penalized a little for not attempting to fix the design
7. If there are flaws in the design, and the coder fixes the design the coder gets additional bonus points for the extra effort.
8. The coder shall EXPLICITLY point out the changes in design.
9. The coder shall not unnecessarily change the design. If this is done, points are taken off from the coder.
10. If design is submitted but code is not submitted or does not work then the designer gets the points for his/her design, the coder does not get any points for his/her code. The coder is classified as a "BAD PERSON".
11. If design is not submitted, or is so bad that it is not worth following then the designer does not get any points for the design. The designer is classified as a "BAD PERSON". The coder then has to do both the design and the code. If the coder does just the code, he/she gets 100 points for the code. If the coder also does a good job on design, then bonus points are given to the coder for the design.
12. If a member is classified as a bad person twice, then on the first chance available, that member is dropped from the team and the good person combined with another good person.
13. If a team member drops the course then the left over member will be combined with an available member. If such a member is not

available, then the remaining member must do both the design and the code.

The team policy ensures that the designer conducts a proper analysis and design and the coder understands and follows the design. Initially, some friction between the team members was observed, but after a while, the members were able to work around their schedules. For larger assignments, parts of the design and code are given by the instructor.

Some amount of class time and lab time is devoted to discussing the assignments. The executable solution of each assignment is made available before the due date. This enables the students to understand the input and output format. The students can also test the performance of their program on certain input data and compare it against the instructors' solution. After the assignment is due, the solution of the assignment is shown to the student, and the design and code are discussed.

The laboratory exercises are conducted in a closed laboratory environment. A lab manual[3] which has exercises based on the text and class material is made available. The objective of each of the exercises is explained first and then the students are allowed to complete the work. The first few lab exercises familiarize the student with the operating system and the Ada compilation environment. Most of the other lab exercises consist of incomplete or incorrect programs which the students have to complete, correct or enhance.

Conclusions

The CS1 course was taught by the author, using the above syllabus, for the first time in Fall 1992. The author has taught the course many times in Pascal and he observed that the software engineering/Ada combination led to better solution designers. Enforcing the completion of design before starting code helped the students understand the solution process much better. They were able to find more flaws in the design. The modular design and development helped them to quickly find problem areas and fix them. The closed lab environment definitely helped the students in reinforcing the concepts learned in the classroom. The number of assignments may be reduced by one or two by combining concepts. The CS2 course based on this approach of CS1 is currently under preparation.

References

1. Hooper, James, *"Planning for Software Engineering Education Within a Computer Science Framework At Marshall University"*, Sixth Software Engineering Institute Conference on Software Engineering Education, Oct 5-7, 1992, San Diego.

2. Lodgher, A., *"Ada Language Design, Style and Documentation Manual"*, Department of CSD, Marshall University.

3. Lodgher, A., *"CS1 - Computer Programming I Lab Manual"*, Department of CSD, Marshall University.

Akhtar Lodgher (Ph.D 1990, George Mason University) is an Assistant Professor in the Department of Computer Science and Software Development (CSD) at Marshall University since Sept 1990. His teaching and research interests are in the fields of software engineering, data structures, algorithms and object oriented programming.

James Hooper (Ph.D 1979, University of Alabama, Birmingham) is a visting Professor, from the University of Alabama, Huntsville, occupying the Arthur and Joan Weisberg Chair in Software Engineering at Marshall University since Fall 1991. His teaching and research interests include software engineering (especially software reuse and the software process), programming languages and discrete event simulation.

A COMPARISON OF ADA AND C AS TEACHING LANGUAGES

Martin L. Barrett
M. Susan Richman
Computer Science Department
Penn State Harrisburg
777 W Harrisburg Pike
Middletown, PA 17057-4898

Abstract

The demands of the marketplace are causing some Computer Science programs to change their main teaching language from Pascal to either Ada or C. This paper discusses the strengths and weaknesses of the two languages in an educational context. The goals of software engineering and general pedagogical concerns are used to structure the discussion. Availability of materials and student attitudes towards the languages are also discussed.

1. Introduction

The choice of a language in a Computer Science program has broad implications for teachers and students. This paper discusses the pros and cons of Ada versus C as a language for computer science education. The context of the paper is the initial core of courses in the curriculum (usually referred to as CS1, CS2, and Data Structures), Both languages offer features that make them possible choices as the support language across the curriculum. The dominance of Pascal as the language of choice in these courses has declined. The main competitors of Pascal for class use are now Ada and C, with some usage of C++, Smalltalk, and Lisp. The marketplace has never used Pascal much, but has consistently supported C and Ada. This has lead to the use of these languages in many schools, either as the main language or at least as an optional language. Both languages have been used for medium and large commercial and research projects.

When evaluating languages for course use, it is important to keep in mind the main goals of a Computer Science curriculum. The ACM/IEEE Curriculum Recommendations[1] offer several alternative program goals. We believe that the most important goals are to produce students who are both *problem solvers* and *communicators*, rather than simply coders. To this end, the software engineering goals and principles are used as a basis for the comparison of the languages.

Some of the issues to be considered when evaluating programming languages for instructional purposes are:

(1.) how difficult is the language to learn/teach?
(2.) does the language encourage good practices and discourage poor practices?
(3.) is the language easy to use?
(4.) does the language inspire or dampen enthusiasm?

In each of these matters, while worthwhile results are often difficult to achieve in computer science education as with other aspects of life, to whatever extent the language assists the student and the instructor, those positive results become more likely to occur.

In addition to discussing features, there is a brief discussion of the availability of texts, compilers, and other resources, and of how the languages are perceived by students.

2. Comparison of Features

Ada and C have many similar features. Simple variable types, control constructs, function and procedure parameters and scoping, and structured data types are different in the details in the two languages, but alike enough for teaching purposes. Important differences exist, however, on a number of issues. Following Booch [2], the goals and principles of software engineering are used to illustrate these differences.

The following four concepts are given by Booch as the goals of software engineering: understandability, reliability, modifiability, and efficiency. The principles of software engineering that allow these goals to be met are discussed below.

Understandability

Quite often, beginning programming students (and advanced students also) enjoy the challenge of communicating in the somewhat alien language understood by computers. If they enjoyed writing clear English prose more than technical jargon, they would probably major in one of the humanistic studies rather than computer science. It's not an accident that the descriptive phrase used is, "playing with the computer."

By using a high-level language which approximates English prose, a student is removed from the alien feeling of having to think at the machine level. Student are given the tools and language to communicate meaning and intentions to whomever is reading and maintaining code. Writing readable code is encouraged.

Ada is sufficiently readable (with careful choice of identifiers) to be mostly self-documenting, comments being used primarily to explain the non-obvious. Comments can be a two-edged sword in code with a long lifetime; if a maintenance programmer does not also modify documentation to meet changes in code, the documentation becomes incorrect.

The sometimes obscure syntax of C can add additional layers of complexity to the problem-solving activity. C code can suffer without discipline by the programmer, since C is not inherently self-documenting. Many examples exist of C statements which even an experienced C programmer would find difficult to decipher. This often involves pointers, arrays, and functions, such as *int *(*f)(int)* To some extent, C programmers consider this kind of code as a badge of honor. Instructors should not encourage this kind of code, of course. Similarly, weak typing in C allows variables to change, chameleon-like, from one type to another at the programmer's whim.

At a higher level, the data structures and modules of a program must be easily understood. If you can't understand the problem and the solution in terms of the problem (i.e. at the abstract level) you can't hope to be able to understand the solution at the implementation level. Abstraction allows students to focus on problem-solving, rather than the details of the implementation.

Both languages have facilities needed to build custom data types and to modularize programs. Logical data types such as linked lists and trees can be built in most structured languages, however, and is not discussed further. Modularity is handled somewhat differently in each language.

For example, C does not allow nesting of functions, while Ada does. In C, a function called only from inside one function is still visible to other functions rather than being private. Ada allows procedures to be nested, so that modularity and need-to-know inforamtion hiding is achieved.

Functions can be grouped into files for separate compilation in both languages. Typically, C function prototypes are placed in a header file that is included by other modules. Ada uses an interface definition for a module. The mechanics are different, but the effect is the same.

C can provide a measure of information hiding through the use of separate compilation modules. Header files provide function prototypes and variable type information to other modules. The details of the data structures themselves can be kept within the modules. Abstract data types are simulated through the use of pointers to data structures and providing only those prototypes necessary to manipulate those data structures. However, since the pointer can be used to access the data structure's components, and the components can be found in the header file, true data abstraction is not possible.

Polymorphic data structures (those that are type-independent) are possible in C by using void pointers. Code that operates on such structures cannot use on any data stored in the structures. This is not enforced by C but relies on programmer discipline.

In Ada, one can easily construct abstract data types using packages, solving the information hiding problem. Only the public interface is accessable by calling routines; private procedures and data types are not accessable. ADT's are one step away from objects.

Polymorphism is handled in a different way in

Ada, using generic packages and instantiation in the appropriate data type.

There are no inherent restrictions on whether the functions in a C compilation module are related to one another. The cohesion of the module is left to the programmer. Likewise, different modules may be related to each other by functionality or by data structures if the programmer chooses, allowing a high degree of coupling. In Ada, packages provide a natural structure for encapsulating logically related types and subprograms. The use of packages encourages logical structuring, but as in C, there is no enforcement of high module cohesiveness or low intermodule coupling. Since these are design issues, an instructor must enforce appropriate standards.

Reliability

Ensuring that a program can prevent and/or recover from errors is a key to producing quality software. Overall software quality assurance is probably beyond the scope of the first few programming courses, but several important issues should be covered.

Type checking can ensure that certain types of errors are prevented. C has weak type checking. For example, an *enum* (enumerated) type is treated as an integer, so that an enumerated variable can be passed in place of an integer parameter. Since C compilers allow this, the *lint* precompiler must be used to detect it. C also lacks the ability to declare subrange types. Out of bounds errors can easily occur, as in array references. Promotion of numeric data types within expressions is often automatic, using a progression from integer to float to double. Manual promotion is also possible, as in *(float)i*.

Ada has strong typing. There are several consequences of this fact: (1.) Safer code. The system catches faulty data and informs student rather than using the data in calculations, making code easier to debug. This requires clear thinking regarding appropriate data types and allowable operations. (2.) More thinking about design and data types is required before coding. (3.) Programs can be more tedious to write because of (2). (4.) Type checking across compilation units assists in building modular code which can be integrated readily. Monolithic code is inherently more difficult to debug.

Exception handling is provided in C by including the *errno* and *signal* header files. Usage of error codes and signals is rather esoteric and is usually omitted from beginning courses.

Exception handling is used frequently in Ada and is quite helpful in finding source of errors. Predefined exceptions are readily caught by the system, and students have options in handling exceptions. Compilers, though, could be more helpful in giving details of conditions under which exceptions were raised.

Modifiability

Modifications to code must have only small, local effects - "controlled change", as Booch says. The key to controlled change is structured, modular design. Almost any modern programming language supports structured programming and modularity. As noted above, C does not allow nesting of functions; all functions are at the same level. Modules - separate compilation units - are supported.

Several factors limit the effects of modularity in C. The fact that global variables are allowed is naturally an inhibitor to controlled change. Variables may be global to a program by placing them in a header file or by declaring them as external, or may be global only to the module that they are declared in.

In Ada, modularity is enhanced by packages and facilities for separate compilation. By developing the specification for a package and delaying consideration of the body's implementation, the student can concentrate on problem-solving. This in turn supports top-down design.

Efficiency

There are several interpretations of efficiency. The one most often used is time related: speed of execution. A second is space related: size of data structures and overall code. The trade-offs between these two are usually discussed in the Data Structures and Algorithms course(s). A third meaning is left to Software Engineering: efficiency of the development cycle. This is related both to the ease of program development and to the ability to reuse code.

Students often become overly concerned with the efficiency of code, sometimes focusing on attempts to achieve marginal increases in speed or

savings in memory at the expense of more important goals such as understandability or modifiability. Recognizing the commonly applicable "90-10 Rule" (90% of execution time is spent in executing 10% of the code) the student can perhaps be persuaded to isolate the expensive 10% of the code, and concentrate on optimizing that without sacrificing other desirable traits of the remaining 90%. The modularity features and adherence to interfaces of Ada units provide powerful support for this approach.

On the other side, sometimes instructors tend to lose sight of the importance of making efficient use of the student's time, energy, and enthusiasm. Programming assignments should ideally be designed so that they challenge, without overwhelming, the student. There should not be an excess of tedious detail to blur the new concepts. The strong typing of Ada often seems tiresome at first, but for understandability and reliability, that is preferable to the weak typing in C which allows variables to change from one type to another. When you violate the rules in Ada, the compiler informs you forthrightly. If you unintentionally violate the rules in C, the compiler proceeds blithely, and you may have no idea that your results are meaningless, or why.

The greater reusability of Ada code encourages and supports the building of modules which can readily be incorporated into larger systems. Such systems can approximate applications in the "real" world more closely than typical programming assignments.

Code reuse in C is accomplished via libraries. Related functions can be gathered together in a module, then compiled and stored as a binary module. C handles many simple tasks by this method rather than including language primitives. There are libraries for mathematics, string operations, and file operations, among others. Users can create their own libraries and often do - for vector functions, image and signal processing, and so on. The amount of reuse of these libraries is low, however, with users forced to write their own libraries if a source cannot be found.

Software components in Ada can be re-used on several levels: (1.) Code can be supplied to a student to use in application. (2.) Students can develop modules to be used as basis for ultimately larger systems. (3.) Maintainable code can be modified (or even made generic) to relieve the tedium of doing "the same stuff with only a few modifications." This is especially helpful in the Data Structures course.

While not appropriate for students to design and write in CS1, simple generic units can be provided for the students to use. This promotes thinking in terms of abstractions and about common properties and algorithms.

3. Instructional Concerns

The difficulty of teaching a language depends, of course, on the audience. Experienced programmers have a context within which to place new ideas; conversely, some old ideas may need to be unlearned. For a beginning language, one would hope to encourage good habits and discourage bad ones.

The basic constructs of each language are quite similar. Variable declarations, looping, decisions, and procedures are handled similarly. A number of differences have been discussed previously; several more are offered here.

Input/output is complicated in Ada by the need to instantiate generic I/O packages. For beginners, the instructor needs to provide an "Easy_IO" package to ease this. C's I/O is more straightforward for beginners. There are simple formatting rules for all types, provided the *stdio* header is included, that allow I/O without too much effort.

Parameter passing in C is complicated by the pointer notation. Rather than Ada's designations of *out* to return values, C requires that a variable's address be passed using the & operator and received using the * de-referencing operator. When combined with the need to pass variables that are already pointers (as in linked list processing), this is quite cumbersome. Ada allows *any* type of variable to be passed - - record, multi-dimensional array, *access* types, private types, or even task types. Access types provide dynamically created variables, similar to C pointers but strongly typed.

While it is possible to write cryptic code in any language, C has the reputation of encouraging such code. Programmer discipline is needed more than in Ada to write clear, concise, modular code.

4. Availability

The availability of Ada texts is no longer the concern it was previously. Introductory texts and data structures texts are now widely available, as are a fair number of Software Engineering books using Ada. The Ada Information Clearinghouse provides a listing of current Ada books (over 130 in the latest list). The Catalog of Resources for Education in Ada and Software Engineering [3], also from the Ada IC, is a list of courses offered in Ada at colleges and universities. Books using C are also readily available. For other courses, though, neither language is commonly used. Ada continues to suffer from the scarcity of affordable, easy to use compilers and PC environments. There are several PC Ada compilers available. There is a freeware Ada interpreter for workstation environments, and a freeware Ada compiler will be available soon. C is normally the default language in workstation environments, so compilers are normally included with the system software. There are a number of popular PC compilers for C that are affordable for students.

5. Student Perspectives

A small group of students who had experience in both Ada and C were interviewed on their views of the languages. Despite the small sample, several interesting points were made.

Overall, Ada programs were easier to read. Programmer discipline, the students realized, was a key to writing readable C programs. This was also true of other goals, such as modularity, tight interfaces, and type checking. In C, most of the students could not overcome their desire to use underlying data type when using *enum* variables, for example. In fact, while Ada's strong typing was seen as an overall advantage, most found it somewhat confining.

There was general agreement that packages and generics were a powerful tool in Ada that C could not match. The resulting modularity of their programs was cited as an important factor in designing programs. In addition, focusing on the procedure/package specifications made it easier to handle information hiding.

Exception handling was superior in Ada. Since it is an inherent part of the language, all the students used it and felt comfortable doing so, realizing that they were writing more reliable code. Input and output were rated as somewhat difficult in Ada, at least for beginners. In C, it was easier to format output data without worrying about instantiating I/O packages. After the students had more experience with generics, this was less of a problem.

C was regarded as a more "powerful" language than Ada. This misperception related to being able to handle low-level programming, such as device manipulation, in C but not in Ada. This kind of programming is, in fact, possible in either language. Pointers were also noted as a powerful feature, but the steep learning curve to manipulate C pointers correctly was mentioned.

6. Conclusions

When comparing programming languages, personal preference plays a large part. In determining what language to use to implement a project, the presence or absence of special features may be critical to efficient implementation, or even to the possibility of implementation. For instruction purposes, though, there are different considerations. This paper presented some ideas relevant to instructional languages from a software engineering framework.

There are several advantages of Ada over C as a teaching language. That C requires programmer discipline to achieve similar effects as Ada shows that Ada is the more natural language for use in instruction. Self-documentation, an emphasis on modularity, and greater reuse of code are all advantages of Ada.

So why choose C? Marketplace pressures ensure its survival, but that is not enough to choose it as a teaching language.

7. References

[1] ACM/IEEE—CS Joint Curriculum Task Force, **Computing Curricula 1991**, Feb. 1991.

[2] Booch, Grady, **Software Engineering With Ada, 2nd Ed.**, Benjamin Cummings Publishing Company, Menlo Park, CA, 1987.

[3] **Catalog of Resources for Education in Ada and Software Engineering**, Ada Information Clearinghouse, Lanham, MD, 1992.

The TIPSE: An educational support environment

M.B. Ratcliffe, B.R.Whittle, M.F. Bott & T.J. Stotter-Brooks
University of Wales, Aberystwyth

Abstract

This paper presents an integrated environment that is under development at the University of Wales, Aberystwyth, with the specific goal of supporting the teaching of software engineering[1]. The environment presents users with a fully integrated tool set that addresses many aspects of the software life cycle. To a large extent, the environment has been developed through the reuse of existing software.

The paper is divided into two parts. First, the technical details are presented This is followed by a discussion of the educational aspects of the environment, its application to a number of different courses, and an evaluation of experiences to date.

1 Introduction

Software Engineering is a common theme running through all of the courses offered by the Computer Science Department at the University of Wales, Aberystwyth; it is evident in the emphasis placed on design, quality assurance and project management. The introduction, in 1986, of Ada as the main programming language has enabled students to apply some of the design principles that they learn, but the lack of specific support software still makes it difficult for them to practise everything they are taught.

The environment being developed by the Software Engineering Research Group at Aberystwyth is capable of supporting most of the software development process. It will eventually provide students with tools to support design, coding, project management, document production, verification, validation and testing.

The environment is far more than a collection of integrated CASE tools [2]. It has been designed to be used in a way which demonstrates many of the concepts of software engineering in a practical and educational manner. For this reason, it is known as the *TIPSE*, an Integrated Project Support Environment for Teaching.

Users of the TIPSE are able to experience first hand the benefits to be gained from using and developing software within a fully integrated environment; an environment that is currently integrated at the level of database and user interface but which will be ultimately integrated at the level of process.

The current release of the TIPSE is already being used by students and its effectiveness evaluated. It is intended that new tools will be added and existing ones enhanced until the TIPSE becomes suitable for use, not only for undergraduate students, but also for participants on advanced software engineering courses. To satisfy the different experience levels of its users, all releases of the TIPSE will have separate

[1] Parts of this paper have previously appeared in an article [1] in the Software Engineering Journal and are reproduced here by kind permission of the editors.

modes of operation for both naive and sophisticated users alike. The system is inherently multi user because of the need to provide effective support for group project work.

2 An integrated approach

The widespread adoption of structured methods by software engineers together with the increased availability of comparatively cheap powerful workstations has lead to a proliferation of CASE tools available on the market. Although there are now tools to assist in almost all aspects of software development, few address more than a very limited number of stages within the software lifecycle. Consequently a typical user will have access to several different CASE tools working on several different platforms. The transfer of data from one tool to another is often difficult and maintaining consistency between the different tools is almost impossible.

The idea of supporting tool interworking is certainly not new. In 1980, the Stoneman Report [3] promoted the idea of using a project database to hold the products of the software development process. Such a database would be all encompassing, storing everything from project plans and initial specifications through to object modules and test data sets. The relationships between the objects would also be captured and tools would only be able to access the information via the database. Over the last ten years, much work has been undertaken in this area particularly by the two international tool support interface projects PCTE [4] and CAIS [5, 6]. The need for a tool support interfaces is now widely accepted and forms the basis for our development of the TIPSE.

Integration at the level of the database is just one facility provided by the TIPSE. Consistency of interaction of all tools with the user is a very powerful integrating principle. Users should be able to move from one tool to another without having to familiarise themselves with alternative methods of interaction. It is not desirable for users to undergo retraining every time a new tool is provided within an environment, indeed this principle should apply not only to different versions of a particular tool but also to completely different tools. Although this is important in all environments, it is particularly relevant from a teaching perspective. Obviously a graphical design editor and an Ada compiler cannot present identical interfaces to the user, but the 'principle of minimum surprise' should hold; in other words, the same user action should have predictably similar effects within different tools. This means, for example, that menu selection should always be done in the same way and that users of the tools should not be presented with different styles of user interface by different tools. MicroSoft Windows 3 is an excellent example of an environment which exploits this principle very effectively.

There is an increasing interest in the role of process modeling as a third axis of integration (see [7] for example). Just how far this is appropriate in a teaching environment is something that we are currently investigating.

3 Foundations of the environment

The Software Engineering Research Group at Aberystwyth first began work in the area of integrated support environments back in 1984 as a partner in the Alvey Eclipse consortium [8]. The second generation IPSE produced by this

consortium has since been developed and exploited commercially by IPSYS Software plc, in the form of the Tool Builder's Kit (TBK) [9]. The influence of a related project, PCTE (Portable Common Tools Environment)[2], can be seen clearly in the development of Eclipse and has been an important consideration within the TIPSE.

Initial ideas about the TIPSE were strengthened by our experiences on the ESPRIT funded DRAGON project [10] in which we developed a prototype structure editor capable of supporting both textual and diagrammatic views of a program from a single underlying data structure.

Throughout our collaborative efforts, one unifying theme has been that of software reuse. It is not surprising therefore that the TIPSE is being developed as far as possible through the reuse of existing components.

At the heart of the TIPSE lies the TBK tool support interface. It consists of libraries of generic facilities which we have used to implement our tools. Tools produced in this way are normal Unix tools which coexist with all other Unix tools. All of these tools present a common user interface and share common database access procedures, so enabling a high degree of integration. Though the current release of the TBK is stand alone, a decision has been made that future releases of the TIPSE will be available on the Emeraude implementation of PCTE.

Fig. 1 illustrates the main component parts of the TIPSE.

Integration through the database

The TBK database at the centre of the TIPSE closely follows the entity-relationship-attribute model of PCTE. Unlike relational database implementations which simply record details of software products held within a conventional file store, the entity model actually stores the objects within the database itself. Specifications, source code objects and even program libraries are all held as individual database entities. The links between these objects, for example those which exist between an object specification and its implementation, are stored as actual links within the database.

A further advantage of adopting the PCTE model is the strong typing that it provides. Through the object management system (OMS), the user is able to define and manipulate objects but only in strict accordance with the rules defined for the particular object types. These rules, that is the properties of the information types, are defined in the form of schema definition sets which are used at run-time to enable visibility over the database. As will be described later, these schema definition sets are fundamental to the provision of multiple views and the support of incrementality within the TIPSE.

As a refinement on top of the PCTE data model, the database provided by TBK permits a fine grain definition of objects in the form of a second tier. In this way it is possible to detail the contents of certain object types; Ada source code might be stored in the form of a syntax tree, for example. Similarly an object at the first tier might be defined as a deliverable document; the second tier definition of such an object then defines the structure of its contents, the breakdown of individual chapters into sections and paragraphs. Facilities for accessing and manipulating the objects at both levels are provided through a single unifying interface.

[2]PCTE has been recently accepted as an ECMA (European Computer Manufacturers Association) standard for CASE tools.

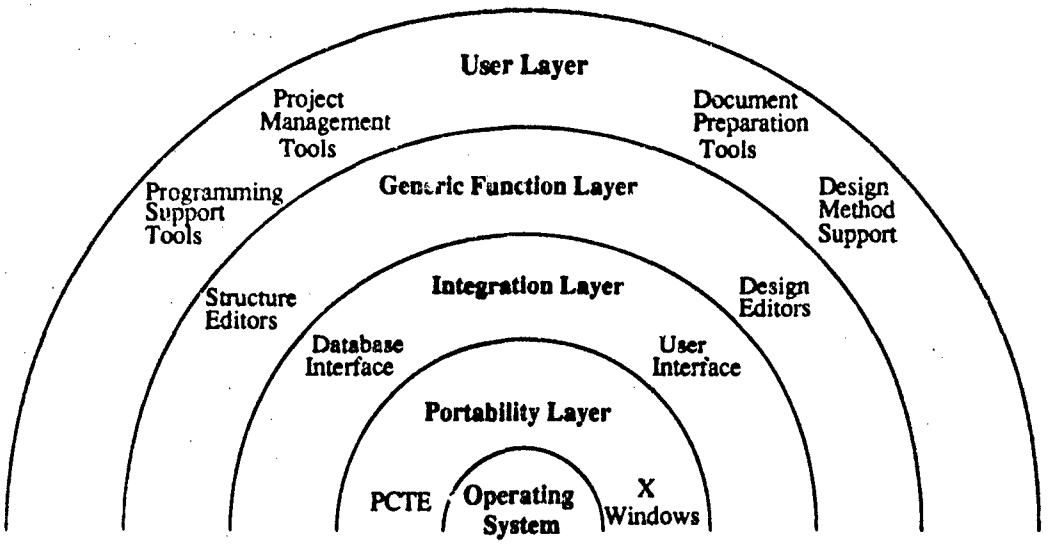


Figure 1: The TIPSE architecture

Integration through the user interface

The second axis of integration within the TIPSE is that of the user interface. All of the tools interact with the user through a set of high level interfaces provided by TBK. These interfaces ensure that a consistent user interface is provided in a device independent way.

The power of the user interface is achieved by separating the front end from the rest of the tool. This idea is based on the philosophy that a tool simply supplies and demands information to and from the user; it need not be concerned with how this information is handled. For example where a tool wishes the user to select a value from a set of possible values, the user might be asked to choose from a menu with all options on display, cycle through a set with one at a time on display, or type a value which is then checked for existence in the permitted set.

To achieve this separation, calls to the interface components are not hard coded into program code as one might expect. Instead, the interaction style and layout of all tools produced using TBK is described by means of the 'Format Definition Language' (FDL). Using this language, the tool builder is able to give a specification detailing layout and functionality of the user interface. FDL does not just define the static appearance of a tool; it is also used to associate operations with the interface, for example, the action to be executed when the user clicks on a button.

The FDL definitions are held in a separate text file which is interpreted on execution of the tool. As the tool starts up, the required user interface is generated as specified by the FDL definitions. If the programmer is unhappy with any aspects of the user interface, the FDL definitions can be easily altered and the tool re-invoked to show the new layout. No recompilation or rebuilding of the tool is necessary. These dynamic aspects of FDL make the user interface facilities of the TIPSE very powerful and enable users within the environment to successfully

develop tool interfaces in a very short period of time.

The high level user interface primitives have been implemented to run under the X Window system [11]. This is a particularly important feature as it enables the environment to include tools which run on hardware configurations other than Sun work stations, the original environment of the TIPSE. The TIPSE may include software which, though running on a PC, interacts with the user sitting at the Sun work station. Such PC applications might be supported by DOS emulators running on the work stations. It is expected that a number of project management tools including Microsoft Project will be supported in this way. Whatever method of implementation is used, as far as the user is concerned, all of the tools will appear to work on a single platform.

4 The meta-tools - building blocks for the TIPSE

The generic function layer of TBK (see fig 1) provides the TIPSE with a number of powerful meta-tools which we have used to generate a closely integrated tool set.

A generic design editor can be instantiated to provide diagramming support for all aspects of software engineering, from simple data diagrams to more complex representations used by a structured method. This editor not only provides operations for manipulating diagrams, but also checks that the design conforms to the rules of the method. With a little specialised customisation by the tool builder a very powerful tool can be produced.

A similar, and complementary tool to the design editor is the generic structure editor. Both tools simply present to the user different representations of objects that are held within the database. The design editor presents a graphical view of these objects; the structure editor presents a textual view. By manipulating the on-screen representation of an object, the user is actually altering the database.

These meta-tools have been used to build tools in a fraction of the time normally required. They form the basis of a number of high level tools many of which are still under development, that will help ensure the student obtains maximum benefit from using such an environment. The tools that are available in the current release of the TIPSE include method support tools (including diagrammatic representation), programming support tools (such as programming language structure editors), version control and basic project management tools.

4.1 The TIPSE front end

The facilities offered by the TIPSE are all accessed through a centralised control panel which has a similar appearance to all of the other tools within the environment. In addition to utilising the user interface primitives, this uniformity of user interface has been achieved by the adoption of a standard interaction metaphor The TBK control panel idea [8, chapter 6], follows the analogy of an interface to a complex piece of hardware, like the operator panel at a power station; the user interface provides interaction possibilities in the form of buttons, menus, state selectors, indicator lights and signs. The similar appearance of the two tools depicted in Figs 2 and 4 demonstrates this consistency.

In support of an incremental philosophy adopted by the TIPSE, the control panel has been designed to offer facilities dependent upon the skill levels of the student. Initially the facilities include only basic editing operations and programming

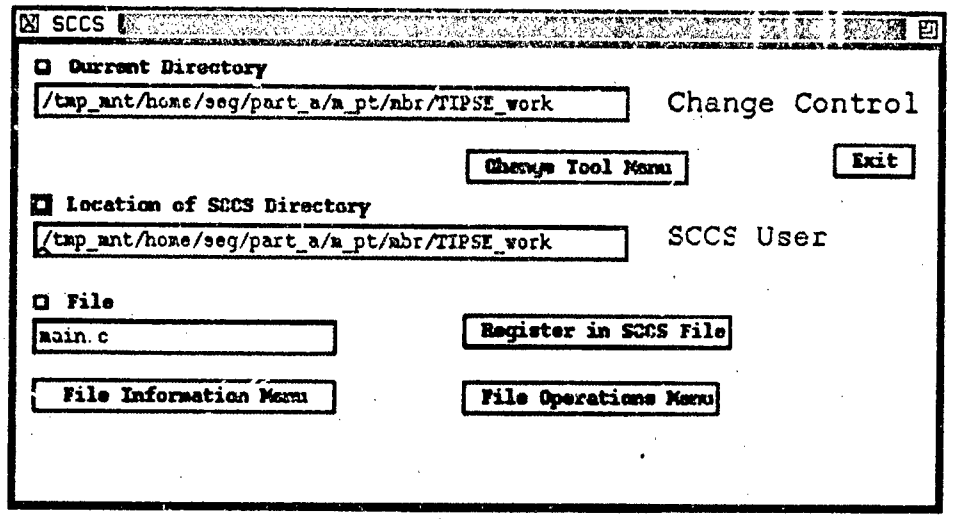


Figure 2: The TIPSE change control tool

support tools, but as the users become more experienced, the available operations are enriched to include configuration control and project management.

The TIPSE as a programming environment

Although the TIPSE now provides tools to support many different stages of the software development process, developments were initially directed at supporting programming. Even before the idea of the TIPSE had been developed, a number of research projects at Aberystwyth were already providing support environments for the Ada programming language.

The most significant of these environments, DDT [12], developed under the DRAGON project [10], supported the notion of multiple views. Though only a prototype, the user of this tool was able to textually specify an Ada program and then, for example, request a graphical view to show the program's relationship with other library units.

The idea of utilising multiple views within a structure editor has been fundamental to the design of our current tool set. Currently Ada is the only supported language, but developments are being planned to support languages such as C and C++. Fig. 3 shows an example invocation of the editor, for a partially developed Ada program.

As with all applications, the schema utilised by the structure editor defines the format of the underlying database and thereby enables other tools to use the data; users are able to invoke a related support tool to obtain HOOD-like graphical views of the Ada programs.

It is anticipated that users of the TIPSE will invoke the design editor to specify the high level design of software systems; at this level a pictorial view is often more beneficial in illustrating both its internal structure and the relationship with other program modules. At any stage the user may switch to a textual view and continue with the specification using the structure editor. Detailed specification in diagrammatic form, down to the level of individual program instructions is not

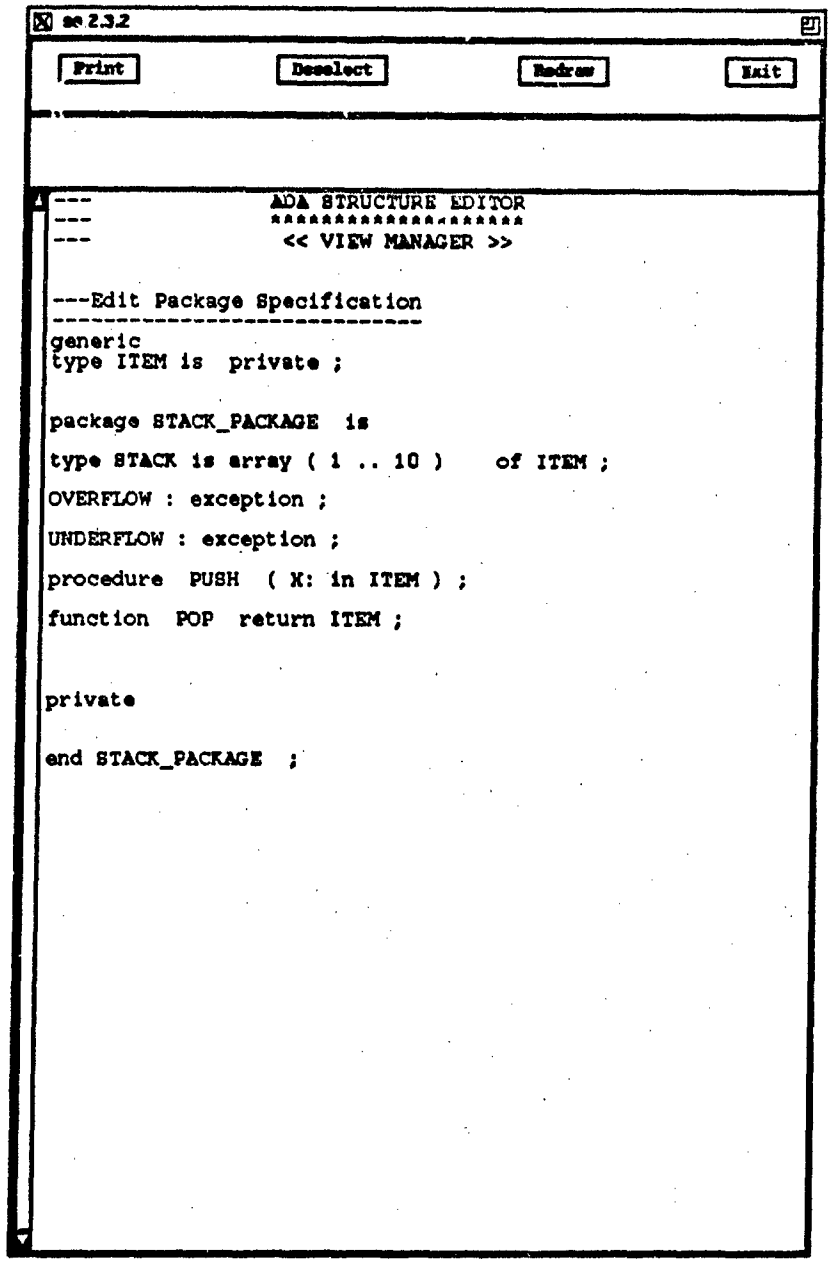


Figure 3: The structure editor: An Ada editor for part of an Ada program

thought to be desirable [13] and will not be supported. The current facilities within the TIPSE have not yet reached a level of functionality to enable evaluation of this approach.

Support for software reuse

During the Eclipse and DRAGON projects, much work was undertaken in the field of software reuse [12, 14]. It should not come as a surprise therefore that the TIPSE places a lot of emphasis on software reuse. This is shown both in terms of the way in which the environment has been developed but also in the support that it provides for the users. Our instantiations of the structure editor and the design editor have been designed to support software reuse and to provide the user with access to a library of reusable components [15]. The available components are compatible with both tools and can be viewed in a textual or graphical mode.

Our experience has been that students working through the TIPSE have found the environment complimentary to the emphasis placed on reuse within the software engineering courses at Aberystwyth. By utilising the library of components, students are better able to appreciate the benefits to be gained from both designing with reuse and designing for reuse.

Design method support

No software engineering course would be complete without discussing the role of design methods.Whilst it is possible to teach the theoretical aspects of a design method within a reasonable period of time, the learning hurdles for the support tools often negates their practical usage. The uniformity of user interfaces within the TIPSE makes it feasible to use a number of design tools at an undergraduate level.

Specific support is currently provided for the HOOD [16] method in the form of the IPSYS HOOD tool set [17]. Used widely by the Aerospace industry, this is a well respected tool kit that supports an increasingly popular method. Based on the underlying TBK facilities, the tools are fully integrated with many aspects of the TIPSE and have very similar front ends. Using this environment, the emphasis of the teaching can be placed on the example method rather than on the complex support tools.

Document Preparation

One of the areas to benefit most from an integrated database should be that of documentation. As traditional environments seldom support any tool interworking, system documentation is often out of step with development. With the facilities of an IPSE, it should be possible to solve this problem. To ensure compatibility, design specifications can be linked into the specification of component interfaces, for example, and textual and diagrammatic specifications can simply be different views of the same data. Initial investigations into this area look promising.

The environment already includes a structure editor to support the development of high quality documents using the LaTeX document preparation system [18]. Furthermore, the adoption of SGML [19] is under consideration. Through the use of templates, this facility should provide the user with a standard structure for the particular type of document under production.

5 The TIPSE as an educational environment

The TIPSE has a rich set of high level tools that are available for its users. The subject of this section is the extent to which the environment can be used to facilitate the educational process.

Initial developments on the TIPSE concentrated on producing high level tools for supporting our undergraduate software engineering courses. These tools would all have graphical front ends to make them more attractive to their users. Facilities such as project management tools, configuration control tools and graphically-based design tools all seemed to be desirable facilities in an educational environment.

As the development of the high level tools proceeded, it became clear that the lower level facilities, the actual building blocks of the environment, were to become very important to the teaching perspectives of the TIPSE. In addition to forming the foundations of many of the higher level tools, the database and user interface facilities have much to offer the trainee software engineer. Within the TIPSE, not only are students able to gain first hand experience of using integrated tools, they can now also use the raw facilities for designing and implementing their own.

Prototyping is now considered to be a very important aspect of software development. The advantages that the technique provides in establishing user requirements, for example, is widely accepted. Our experience of using the TIPSE for advanced software engineering courses has been that the user interface facilities are very well suited to rapid prototyping. After an initial learning period, we have found that students can very quickly produce a fairly sophisticated front end. Designing user interfaces is notoriously difficult and can involve a lot of tinkering; fortunately the facilities provided within the TIPSE make this task a lot easier. With the growing realisation of the importance of designing user friendly interfaces, this area is likely to become far more significant

The database is equally as important as a stand alone facility. Students are instructed on the Object Management System and how the structure of data to be manipulated is defined through the use of a data definition language, detailing the types of objects, links and attributes which can occur within the database. They are able to investigate how the use of a schema grants visibility to the object types it defines, instances of which can then be created, manipulated and deleted. Using a high level language, designed to simplify the task, students are able to experiment in defining schemas and in populating the database.

Fig. 4 shows a simple management tool that was initially developed as a student project. The tool provides some basic facilities for interrogating and updating the database and has been enhanced with the addition of a simple front end developed using FDL. Other student projects have lead to the development of more sophisticated front ends to the model using the generic tools; design editors and structure editors now provide a complete suite of tools.

One of the main educational benefits of the TIPSE is its open architecture. New tools can be added to the environment becoming fully integrated through the sharing of common schemas. To support this facility, schemas utilised by the predefined TIPSE tools are also available to our student users thus enabling them to write new tools to interact with the database. Modifications can be made to database objects and the

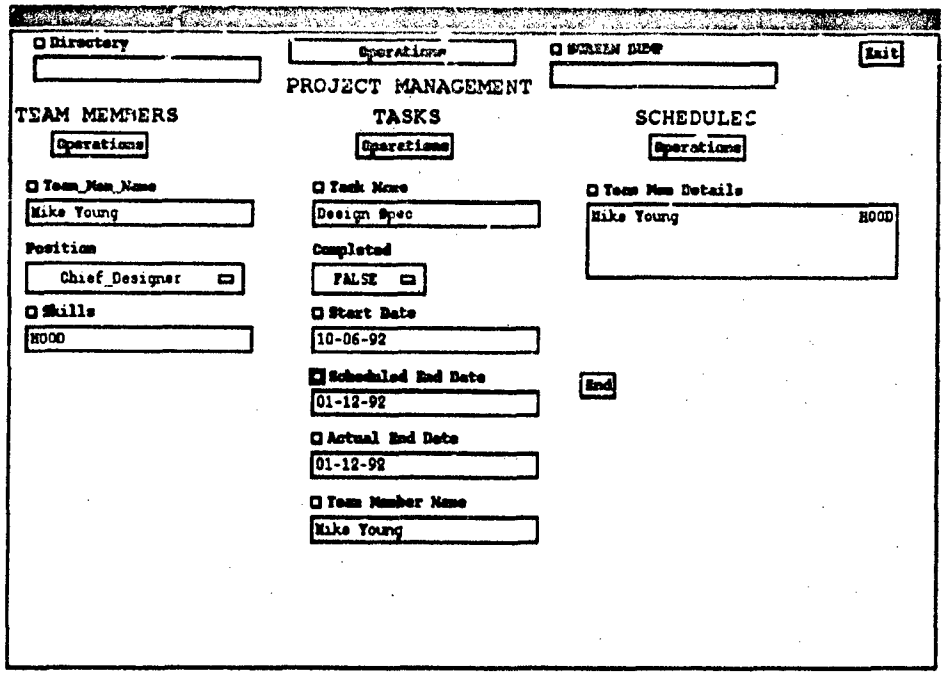


Figure 4: Project management tool

effects investigated by invoking one of the predefined tools. This kind of exercise has largely been confined to our more advanced software engineering courses but the results have been very positive.

5.1 Incrementality

Central to the learning process of the TIPSE has been the idea of incrementality. Whilst the open architecture will enable the TIPSE to change through time and thus be extensible, it will also develop with its users and will change in accordance with their level of knowledge and skill. This quality can be achieved in many ways; a less complex tool could be replaced with a more advanced tool, for example, the exchange of a textual editor for a structured editor. Alternatively a tool could be enriched by adding to its functionality; a structure editor that supports only a small subset of the language could be replaced by one supporting a larger subset. A more subtle example might be the introduction of explicit version control features.

Given our commitment to the principle of minimal surprise, it is important that an incremental change to a tool does not invalidate what has gone before; in particular, the same way of manipulating the user interface should produce the same results and the incremented version should continue to operate successfully on a database developed under the previous version.

In the class room, students are gradually introduced to the structures of the language through a logical progression of examples. These examples, in programming terms, explain not only the structure and syntax of a language, but also its semantic logic. It is our intention to retain this paradigm, primarily through the use of a family of structure editors, a decision

motivated by our experience on other research projects, such as [12]. Through the use of these editors, the students are presented with an environment in which the structures they have studied appear to be the only structures in the language. The user is freed from the syntax details of the new structures, whilst at the same time being insulated from the other more complex language constructs which are yet to be learned.

The incremental concepts pose an extra burden on compatibility within the environment where, for the purposes of teaching, it is important that the previous work of students is available at a later date so that they may learn from their mistakes or reuse their earlier code. Fortunately the schema facilities of the database and the generic tools available from within TBK have made this task much easier.

Incremental editors within the TIPSE

The TIPSE provides a family of structure editors, all of which are instantiations of the generic structure editor, driven by an abstract syntax. This abstract syntax represents the schema for the underlying data structure and is the same for all editors in the family. Incrementation is achieved by changing the concrete syntax, which specifies what an individual editor may create and how it views the database; multiple views both textual and diagrammatic, are specified in this way. The system is therefore being implemented such that although the underlying representation is constant, the user view of the structure can change.

To control the incrementation, the TIPSE allows students to choose the level at which they will work, within limits set by the lecturer. The levels are numbered and the lecturer can specify that all students within a given group may use levels 3 to 5; this means that, when students invoke the structure editor, they will be offered this choice of levels, with a brief indication of what each provides. As the course progresses, the lecturer may change the range to allow levels 5 to 8, and so on.

5.2 Applying the TIPSE to undergraduate courses

The TIPSE has been designed to be applicable to software engineers with very different levels of experience. Ultimately, it is our intention to use the TIPSE as a practical basis for a number of external courses run for industry. Until it gains full functionality, the the environment is currently restricted to undergraduate use.

In the first year of their degree course, students take an introductory course where they learn to program in Ada and become familiar with basic data structures; emphasis is also placed on introducing the key concepts of software engineering. During this period, through a process of continual assessment, students carry out a small number of individual projects. The programming complexity is usually low, for example, a menu driven temperature conversion system, or a simple line based editor. However, even at this stage, students are expected to follow a well engineered approach to the development of their software.

The main use of the TIPSE is at the level of the integrated front end which provides access to tools for design, coding and execution of Ada programs. The incremental nature of the structure editor closely complements the teaching of Ada, and should be well suited to an introductory course.

Through the second year, the student continues to study and write Ada, a

further main application of the TIPSE is the support that it gives for group project work, an important element of the software engineering course. The actual programming tasks are not too complicated; the role of this exercice is to focus the students attention on the group and the interaction of group working. The project groups typically contain six students, who liaise with a client, a staff member who also acts as a consultant/advisor, to develop a project specification and follow this specification through to implementation, delivery and acceptance testing. The facilities that the TIPSE provides for project management, configuration management and project specification nicely complement the material presented in lectures. Certainly, the ease with which the tools can be used, compared to the command line interface or hand techniques, encourages the students to utilise the tools.

Final year students attend a course on advanced software engineering, where they are instructed in the principles of IPSE technology, the development of meta tools and support for programming in the large. As part of the practical element of this course the students are encouraged to use the TIPSE as an example IPSE. Rather like the medical student's skeleton, the student can learn principles from the simple structure presented by the TIPSE, by exploring the make up of an environment with which they are already familiar. Being familiar with what a tool can do, the students can concentrate on how the tool is implemented. A typical project may involve prototyping a user interface or manipulating the database; such examples were described earlier in this paper.

The scope of applying the TIPSE to undergraduate courses is not limited by the currently available toolset, its extensible nature makes it an ideal candidate for the final year projects when many students produce additional tools for the environment. The knowledge that their tool will actually be used in the future by other students provides many with the incentive to produce a robust tool when they might otherwise only produce a prototype. The use of TIPSE based facilities will expand courses throughout the degree programme as appropriate tools are developed to support them.

5.3 Evaluating the TIPSE

In order to carry out a reasonable evaluation of any tool, it is important to establish a set of criteria under which the evaluation is to be carried cut. In industry such an evaluation might be carried out at two levels: first from the point of user satisfaction and second from a management perspective. Are the users happy with the tool, and do they they feel it is effective? Are the managers satisfied that the tool has provided the appropriate gains to justify the investment? In education a similar evaluation might be carried out. with the students as users and the lecturers as the managers. Student satisfaction is relatively straight forward to measure. If they use a tool after it has been introduced and then continue to use it when any associated assignments have been completed, then one might reasonably assume that they are satisfied. Certainly students rarely need encouragement in voicing their opinions.

The educational benefit gained from using a tool is rather harder to ascertain. The difficulty lies in establishing a control. To request that a particular group of students use manual techniques while their peers are instructed on a fully integrated graphical tool seems unreasonable. Moreover to be effective, such an approach might also

require a different emphasis in associated lectures.

In conclusion, the evaluation of the TIPSE has largely been one of hear say. The environment is still in its infancy and suitable techniques for more scientific evaluation are still being sought.

6 Conclusion

This paper has attempted to give a flavour of the tools provided within the TIPSE, rather than to detail the method of implementation. More specific information is described in [1].

A university education consists of a broadening as well as a deepening of knowledge; at Aberystwyth emphasis is placed on rigorous software engineering principles, it is hoped that the TIPSE will support and enhance the perception of these principles. It could appear that the TIPSE provides a restrictive and over protected environment that does not equipe students for the situation which they may subsequently face in industry. We feel that the main benefit arises when the TIPSE is used as a support tool, to reinforce the principles of the lectures rather than to give emphasis to a particular method or technique.

Though the TIPSE has now been under development for a number of years, this has largely been through the efforts of postgraduate students. Only in recent months has full time effort been provided on the project. As a consequence of this method of development, the project is continuing to follow a phased approach, with the software being released in several distinct stages. At each stage the functionality is enhanced and the reaction of students assessed. The current release of the TIPSE provides many powerful facilities and is the subject of evaluation by both undergraduate and postgraduate users.

7 Acknowledgments

The work described in this paper is a development of work carried out on earlier projects. We are grateful to the Alvey Directorate for its support of the Eclipse programme and to the European Commission for its support of the DRAGON and Sapphire projects; in that context we must also acknowledge the contributions of our many collaborators.

The work on the TIPSE itself would not have been possible without the support of IPSYS Software plc; not only have they provided and supported the TBK software but they have maintained a continuing interest in the work and contributed many useful ideas.

Mr. Whittle's work on the TIPSE was supported by Research Studentship No. 90302773 awarded by the Science and Engineering Research Council.

References

[1] M.B. Ratcliffe, M.F. Bott, T.J. Stotter-Brooks, and B.R. Whittle. The TIPSE: An IPSE for teaching. *Software Engineering Journal.*, September 1992.

[2] A.S. Fisher. *CASE, using software development tools.* John Wiley and Sons Ltd., Chichester, UK, 1988.

[3] J. N. Buxton. *STONEMAN: Requirements for Ada Programming Support Environments.* United States Department of Defense, Washington, 1980.

[4] Commission of the European Communities, DG XIII/Esprit, 45 Av.

Auderghem, Brussels. *PCTE : A Basis for a Portable Common Tool Environment*, Functional Specification, 1.5 edition, 1989.

[5] Washington United States Department of Defense. Military Standard DOD-STD-1838 Common APSE Interface Set (CAIS), October 1986 (actually published in 1987).

[6] Washington
United States Department of Defense. *Military Standard DOD-STD-1838A Common APSE Interface Set (CAIS) (Revision A)*, April 1989.

[7] A. I. Wasserman. Tool integration in software engineering environments. In *Software Engineering Environments.* Lecture Notes in Computer Science, 467, Springer Verlag, 1989.

[8] M.F.Bott, editor. *Eclipse: An Integrated Project Support Environment.* IEE computing series 14. Peter Peregrinus Ltd., on behalf of the IEE, 1989.

[9] IPSYS Software plc, Marlborough Court, Pickford St., Macclesfield, Cheshire, UK. *IPSYS Tool Builder's Kit*, 1990.

[10] A. Di Maio, I. Sommerville, R. Bayan, M. F. Bott, and M. Wirsing. The DRAGON Project. In *Proceedings of ESPRIT '89 Conference*, London, 1989. Kluwer Academic Publishers.

[11] O. Jones. *Introduction to the X window system.* Prentice Hall, New Jersey, USA, 1989.

[12] M. B. Ratcliffe, C. Wang, P. J. Gautier, and B. R. Whittle. Dora - a structure oriented environment generator. *Software Engineering Journal*, 7(3), May 1992.

[13] M. B. Ratcliffe. *Prototyping through the Reuse of Existing Components.* PhD thesis, University of Wales, 1989.

[14] M. B. Ratcliffe and R. J. Gautier. System development through the reuse of existing components. *Software Engineering Journal*, November 1991.

[15] R.J. Gautier and P.J.L. Wallis. *Software Reuse with Ada.* Peter Peregrinus, UK, 1990.

[16] European Space Agency, HOOD Working Group, Issue 3.1. *HOOD Reference Manual*, 1990.

[17] IPSYS Software plc, Marlborough Court, Pickford St, Macclesfield, Cheshire, UK. *IPSYS HOOD tool kit*, 1990.

[18] L. Lamport. LaTeX: *A Document Preparation System.* Addison Wesley, Wokingham, UK., 1986.

[19] B. Martin. *SGML: an author's guide to the Standard Generalized Markup Language.* Addison-Wesley, Wokingham, UK., 1988.

Dr. M. Ratcliffe

After graduating in Computer Science from the University of Wales, Aberystwyth in 1984, Mark Ratcliffe joined the staff as a Research Associate working on the Alvey funded Eclipse Project and the ESPRIT funded Dragon project. During this period, he carried out research in the field of Software Reuse, the area in which he subsequently gained his Ph.D. As a lecturer at Aberystwyth since 1989, his main interests lie in the area of software development environments with specific emphasis on reuse. Over the last few years he has presented, a number of industrial training courses on integrated environments, specifically PCTE and the IPSYS TBK. During the academic year 1992-93, Dr. Ratcliffe is working at the University of Puget Sound, Washington as part of an academic exchange programme.

Email: mbr@aber.ac.uk

B.R. Whittle

Ben Whittle graduated in Agricultural Economics from UW Aberystwyth in 1989. He subsequently completed a masters in Computer Science and was invited to proceed to research for a Ph.D. in Software Engineering. As well as his close interest in the work of the TIPSE project, Mr Whittle is also interested in Software Reuse. He is a member of the British Computer Society Reuse special interest group committee and the editor of the group newsletter.

Email: brw@aber.ac.uk

M.F. Bott

Frank Bott was educated at Wolverhampton Grammar School and Trinity College, Cambridge, where he graduated in Mathematics in 1962. After graduating he worked in the Cambridge University Computing Laboratory until he joined SPL International in 1968. He returned to the academic world in 1977, taking up a visiting appointment at the university of Missouri to assist in establishing a Computer Science programme. In 1979, Mr. Bott joined the Computer Science Department at the University of Wales, Aberystwyth, of which he is now Head. Since being in Aberystwyth he has run a number or research projects funded by the ALVEY programme and the European Commissions ESPRIT and RACE programmes. He is also interested in software engineering education and has edited a recent special issue of the Software Engineering Journal on this topic. He is a member of the British Computer Society and of its Professional Board.

Email: mfb@aber.ac.uk

T.J. Stotter-Brooks

Tim Stotter-Brooks graduated with a B.Sc. in Economic History from the University of Birmingham in 1989. He completed an M.Sc. in Computer Science at UW Aberystwyth, before becoming a research assistant on the TIPSE project in 1991. Mr. Stotter-Brooks is currently a research assistant on the RACE 2076 BOOST project at Aberystwyth.

Email: tjs@aber.ac.uk

All of the authors can be reached through the Dept. of Computer Science, UW Aberystwyth, Aberystwyth, Dyfed, Wales, UK.

The Rapid Development Methodology Applied to Software Intensive Projects

Lynn G. Gref
Jet Propulsion Laboratory
Pasadena, California

William H. Spuck III
Jet Propulsion Laboratory
Pasadena, California

The Rapid Development Methodology (RDM) is an alternative means of developing systems to that of the conventional waterfall method. RDM has been developed independently at the Jet Propulsion Laboratory (JPL) but is consistent with the objectives of the Evolutionary Development approach articulated by the DoD. Ada with its modern software engineering features has proven to be complimentary to RDM. RDM derives from rapid prototyping but is distinct from it. Basic to RDM is the delivery of useful operational increments of the system to a using organization every nine to fifteen months. Each incremental delivery builds on the previous ones and is part of the final delivered system. Documentation and other integrated logistics support items of a formal system development are evolved so in the end the delivered system under RDM is indistinguishable from that developed under the conventional method. Significant advantages of RDM include the satisfaction of true user needs, delivered system functional capabilities during the tenure of sponsors and users, and a process that is adaptive to changing funding profiles and user requirements. RDM has been used at JPL primarily for the development of large software intensive systems with over 30 incremental deliveries having been made. Project sizes have ranged from $10 million spanning a few years to $100 million over a period of 10 years.

Introduction

This paper is about a better, faster, cheaper approach to implementing software-intensive systems, that has been pioneered and refined by the Jet Propulsion Laboratory (JPL). This new approach, the "Rapid Development Methodology" (RDM), has been successfully employed in the development of Ada-based systems, though it can be used with any software language. RDM is an outgrowth of rapid prototyping concepts and is a refinement of the evolutionary acquisition model.

RDM employs incremental development and fielding of the system. User experience with the currently fielded increment of the system provides a basis for the detailed requirements of future deliveries. This assures user satisfaction at the end of the developmental cycle. Funding changes and uncertainties, such as those depicted in Figure 1, are accommodated by fixing the current delivery (e.g. specifications, budgets and schedules) and making necessary adjustments in future incremental deliveries.

Similarly, RDM accommodates evolving or even radical changes in the roles and missions of system users. In the case of one Command Center System the mission has evolved from a focus on a tactical engagement with the Soviet Union to one including humanitarian relief and monitoring national unrest.

Each incremental delivery is treated essentially as a "complete end-to-end requirements definition, implementation and deployment cycle." This permits evolution of the system infrastructure and its functionality, including achieving compliance with new and evolving standards (e.g. POSIX, X-Windows, MOTIF, GOSIP, computer security standards) as commercial, standards-compliant products become available.

New technologies can likewise be incorporated into the system. In one case, a "main frame" database management system server was replaced

by a RISC-based "workstation" to achieve better system performance and transportability at less cost than the annual maintenance costs of the old system. Another example is the enhancement of the Local Area Network (LAN) with an FDDI-based backbone in the fourth incremental delivery.

RDM and Ada have proven to be highly complementary. The objectives of the Ada programming language are to: 1) establish a common programming language, 2) embody and enforce modern software engineering principles, and 3) facilitate the transfer of software to different hardware platforms and operating systems. Under RDM, Ada's embodiment and enforcement of principles such as abstraction, information hiding, program modularity, localization, uniformity, completeness and confirmability directly supports the evolutionary growth of the delivered system. Independence from reliance on specific hardware platforms and operating systems facilitates the insertion of new technology into the system. In fact, RDM achieves the "desired" benefits of Ada within the development lifetime of a single software intensive system.

RDM provides a mechanism for faster and cheaper development. First, the important operational needs of the user are met through a prioritization that determines the content for each incremental delivery. This avoids the cost and time of classical methods which require the development of "every conceivable capability that can be imagined" to avoid discovering omissions after system delivery.

Successful application of RDM strongly suggests adherence to the 80-20 rule. (The 80-20 rule states that it takes 80% of the total effort to achieve the last 20% of the goal.) Under RDM it is possible to set and achieve targets for each incremental delivery that are something less than the ultimate goal (i.e., 80%). After just a few deliveries, the essential portion of the goal is achieved with only a fraction of the effort required by the old, "single thrust" method. Additionally, some "project management items" will be eliminated, since each incremental delivery takes approximately one year from start to finish. For example, there is neither schedule time nor need for all the formal reviews of the conventional development method.

Experience with RDM has lead to the practice of "just in time" engineering. That is, do only essential tasks and put off non-essentials. This focuses the efforts of the staff and eliminates unneeded work. Documentation is handled on a "when" and "as needed" basis, with review comments incorporated into the next delivery.

JPL's experience shows that a single incremental delivery implements somewhere between a third and half the capability resulting from a normal five year conventional development cycle, at the same level of effort. Experience has shown that RDM allows more effort to be channeled into implementation of capabilities.

Description of RDM

First, RDM is neither rapid prototyping nor a version of the spiral incremental development model. Rapid prototyping is used to validate either requirements or design approaches. When completed, the prototype is generally abandoned and the "real" system implemented. Under the spiral model a system is developed in increments and deployed at the end of the development. Each increment progresses through requirements analysis, design, implementation, test and user validation. Hence, the common reference made to it is "build a little, test a little."

Contrary to rapid prototyping, RDM is used to implement systems. The intent

of RDM is to make use of every product. Experience has shown there is some waste attributable to several factors. Use of the system results in changes to the users' operational procedures, with subsequent modifications to system requirements. These lead to design changes and "breakage" of the delivered capabilities. Besides basic system implementation, RDM incrementally develops the system Integrated Logistics Support(ILS) into a full capability at the conclusion of the project. The ILS capability provided at each delivery is tailored to meet the needs of that delivery taking into account that the developers provide most of the support until the system is finally turned over to the government for operations and maintenance.

As opposed to the spiral model, RDM delivers each increment of the system into immediate operational use. As a result, a system developed under RDM is fully operationally tested and validated prior to turnover to the government for sustainment.

RDM is a specific project management approach. It has a set of underlying tenets. It prescribes management policies and procedures for system implementation issues such as project planning, systems engineering, configuration management and documentation. The four tenets of RDM are:

1. Build and deliver the system in a series of regular and consecutive increments.

2. Actively obtain feedback from actual field usage and incorporate the feedback into the system requirements of future deliveries.

3. Involve the users in extensive interaction throughout the development cycle.

4. Implement the system with progressive formality so as to achieve everything essential to sustainment upon delivery of the final increment.

High level planning is done during an initial project definition phase, when a consensus between the developer and customer is reached on a target final operational capability (FOC) requirements specification, a system architecture, an overall budget and an overall schedule. A modular and flexible architecture is essential to support the incremental delivery of the system and the evolutionary specification of system requirements. Modern distributed information system architectures provide one class of examples suitable to RDM's incremental deliveries.

An overall schedule for a generic project is depicted in Figure 2. As discussed, the project under RDM consists of a series of deliveries. Each delivery must go through a "mini" life cycle of its own consisting of "mini" phases. These echo the phases of a conventional development life cycle with the numbers corresponding to the phases: (1) Planning, (2) Requirements, (3) Design, (4) Implementation, (5) Integration and test, (6) Installation, (7) Certification, and (8) Operations and sustainment.

As depicted in Figure 2, the overall planning (phase 1) is done once at the beginning of the project. This plan is updated and modified whenever necessary (e.g. overall budget changes). The FOC Requirements Specification is updated as the system evolves and users learn more what their real needs are. Each delivery begins with a mini-planning effort which determines the specific scope of the delivery (i.e. system functional requirements), the budget and the schedule. Phases two through seven are repeated once for each delivery. And of course, the delivered system is operated and sustained continuously after the first delivery. Once a delivery has been transferred to

operations, the previous delivery has been superseded and it disappears. It is often possible for the Project Definition to overlap the first one or two deliveries, frequently causing them to be called "preliminary deliveries" in recognition of the lack of a complete long-term perspective. These preliminary deliveries have characteristically consisted of capabilities which have been recognized as essential and have been delivered previously in other projects.

Experience has shown that the time interval between successive deliveries (i.e. transfers of capability to user operations) should fall somewhere between nine and fifteen months. A series of deliveries with intervals less than nine months have proven to be impossible to sustain. This is due, in part, to the effort of the development staff needed to sustain (e.g. fix problems) the previous delivery. Also, "too little implementation time" leads to a temptation to "cut corners" by compromising other phases (e.g. integration and test). Intervals greater than fifteen months begin to lose the characteristics of RDM. External influences (e.g. funding changes, requirements changes) begin to upset the desired stability of the current delivery. A delivery cycle that fits with the government's budgetary cycle permits securing full funding of the delivery prior to the commitment to the delivery. Thus the delivery can be made as planned without budgetary influence. Changes in future funding impacts only future deliveries.

Individual incremental deliveries need to be sufficiently small in cost. scope and complexity to forego the safeguards, reviews and formality associated with the conventional development methods. Otherwise, the five to eight year conventional development cycle cannot be broken. (18-24 months are required for the formal reviews. A similar amount of time is required for documentation.)

Five formal reviews have been identified under RDM. A one time Project Definition Review at the end of the Project Definition Phase assesses overall project plans and processes. It gains the concurrence of all parties to the project. At the beginning of the delivery cycle the sponsor's Configuration Control Board holds a Requirements Review to baseline the requirements of the current delivery. This review achieves concurrence among all parties as to the established requirements for the delivery. A Delivery Commitment Review is held early into each delivery cycle to demonstrate understanding of the requirements and general readiness to complete the delivery. This review covers system requirements, system design, implementation plans, costs, schedules and risks. Near the end of each delivery cycle a Delivery Pre-Ship Review is held to establish readiness to undertake system installation, integration and test at the user's site. Long lead time installation items may be shipped and begun installation prior to the Delivery Pre-Ship Review. This review covers design and implementation status, testing thoroughness and results, plans for testing training, operations, maintenance, costs and schedules. The delivery culminates with a Delivery Transfer to Operations Review which is held by the sponsor's Configuration Control Board. This review serves as a system acceptance review for the sponsor and user.

The RDM approach to documentation emphasizes supporting usage and sustainment of the system. Documents evolve with increasing content as the system evolves. Documents are delivered as they become available or just in time of need. They reflect the "as built" system. Document re-work within a delivery cycle is minimized. Precise document suites are tailored by the specific needs of the project.

Project planning documents may include: Project Plan, Hardware Management Plan, Software Management Plan, Safety Plan, Product Assurance Plan, Integrated Logistics Plan, Review Plan, Documentation Plan, Configuration Management Plan, Integration and Testing Plan, Security Plan and Shipping Plan.

Documents delivered within each delivery may include: Requirements Specification, Integrated Support Plan, Segment Requirements Specification, Segment Design Document, Interface Control Document, Database Design Document, Site Concurrence Memorandum, Integrated Test Plans and Procedures, Test/Analysis Report, Security Evaluation Report, Installation Drawings Package, Release Description Document, User Manuals and Training Materials. Figure 3 indicates the progression of the formality of the delivery dependent documents.

Testing under RDM emphasizes satisfying the user. Its key outcome is problem identification. Results of testing are used to determine what is actually included in the delivery. That is, if testing has shown that some segment of the delivery is not operationally viable then its delivery is postponed until a later date or until the next delivery. Therefore, all deliveries are made on time, albeit they may not provide the full functionality as originally planned.

A major testing principle is that to continue to test a system within a single environment using a fixed set of procedures eventually produces diminishing returns as fewer and fewer problems are identified. Exhaustive testing is recognized as impossible to achieve. Once the problem time curve begins to level off the efficacy of testing is increased by changing the focus or the environment. This is depicted in Figure 4.

The initial focus is on integration testing employing data flow threads as a basis for test procedures. Once the system is working as a unit the testing emphasis may be shifted to requirements testing to assure that all the individual functional capabilities work as expected. Another environment is generated with tests that emphasize operational strings or scenarios that emulate the anticipated use of the system. Changing the location of the testing from the laboratory to the users' site where each set of previously used test procedures are repeated creates further test environments. Security testing, acceptance testing and user training provide additional environments.

Finally, the use of the system for operations by real users provides the final test environments. For this reason, the first users must be considered as part of the test team. No amount of formal testing is going to eliminate the first users uncovering problems. Testing needs to minimize the number of problems found by users. However, real use is essential to uncover performance problems, system weaknesses, bottlenecks, and strain points. On this basis the use of the system for another application or radically different operation will constitute a new environment and will undoubtedly result in a flux of previously unidentified problems.

Ada Considerations

Experience with Ada based projects using RDM has shown that 45% of the implementation time in each incremental delivery is spent on preliminary and detailed design activities. Coding and unit testing constitutes 25% of the time and integration and test takes the remaining 30%. This contrasts with the distribution of 20%, 55% and 25%, respectively, that typically has been experienced with the conventional development process.

The greater time for design has been attributed to the additional effort required to define all data types and develop package specifications. Ada, being a strongly typed language, requires more time to define and negotiate all required data structures and type definitions. Less time is required for coding under Ada and RDM because most sub-program interfaces are negotiated and agreed upon during the design phase. This carries over to testing where very little test time is spent dealing with data type errors and sub-program interface mismatches. The rigid specification of software program elements (i.e. procedure argument lists, data types and external package references) minimizes problems with their usage by the various members of a software development staff.

An Ada development environment has proven most beneficial in conjunction with RDM. The Rational Ada Development System provides a 2167A document generator, configuration management system, language sensitive editor and Ada compiler. The development environment assists the developer with views of data type definitions, interdependency information, syntactic and semantic assistance, and incremental compilation support. Code is first generated and compiled on the Rational. After achieving a successful compile the code is transferred to the target environment for unit testing. This transfer is managed automatically by the Rational.

RDM necessitates managing several configuration baselines. The current deployed baseline and the current development baseline are necessary. Frequently, an update to the deployed baseline will be in the works, as well. Maintaining previous operational incremental delivery baselines constitutes good practice. The Rational provides good automated support to the essential configuration management function.

Several features of the Ada language have been exploited in conjunction with RDM. Taking advantage of the inherent readability of Ada code, emphasis has been placed on developing Ada package specifications containing actual Ada code, rather than pseudo-code. This avoids the step of translating the pseudo-code into actual Ada code. Also, interdependencies of the compilation units are rigorously defined. Thus, any undesirable dependencies or architectural problems can be identified and corrected in the design phase rather than later during coding or integration. Ada provides extensive error checking during compilation. This avoids having to find many errors during unit testing on the target system. Finally, extensive use of modularity and common libraries permits significant code re-use from delivery to delivery. This provides the ability to extensively rework applications with a minimum of code change.

Summary

Where the tenets fit, RDM has proven to be far superior to the conventional developmental methodology. RDM is flexible and responsive to the exigencies of real projects. RDM is essentially a design-to-cost process in which users, sponsors, and developers must reach consensus on priorities and scope for each incremental delivery. The result should be the best capability available for the available funding -- financially efficient and effective.

RDM's has been shown to be highly responsive to programmatic and technical changes. Even the obsolescence of system elements during the development life cycle can be accommodated with replacements during the succession of deliveries. RDM is

responsive to the user through its involvement of the user throughout the development cycle. The overt attention on requirements feedback based on operational use assures user satisfaction.

RDM and Ada have proven to be compatible for the development of software intensive systems. Ada's modern software engineering features facilitate the short implementation period of the incremental deliveries essential to RDM. Use of an Ada development environment has contributed to the successful employment of RDM.

JPL has used RDM on seven software intensive system development projects to make 30 successful incremental deliveries. These projects include command centers for the USAF Military Airlift Command, US Transportation Command, US European Command and US Army Headquarters. RDM has also been used to develop war game simulations for the US Army.

So far, JPL has been the sole practitioner of RDM. It remains to be seen how RDM can be used within the government's competitive acquisition guidelines. A task work order contract may be a potential viable contractual candidate. Still, it would appear that changes in Military Standards are necessary to recognize development methods such a RDM. These include the areas of reviews, documentation and testing.

Lynn G. Gref
Manager, Defense Program
Jet Propulsion Laboratory
Mail Stop 79-6
4800 Oak Grove Drive
Pasadena, California 91109-8099

Dr. Gref has in excess of 25 years experience in the systems engineering and development of systems for the Department of Defense and other government agencies. He has managed the implementation of several software intensive projects employing conventional, prototyping and rapid development methodologies. He has been a contributor to the evolution and articulation of RDM at the Jet Propulsion Laboratory.

William H. Spuck III
Manager, Commercial & Civil Programs
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, California 91109-8099

Dr. Spuck has in excess of 30 years experience in systems engineering and program management. All of the systems developed using RDM at JPL have been under Dr. Spuck as the Program Manager. He has provided direction and articulation to the development of the rapid development methodology.

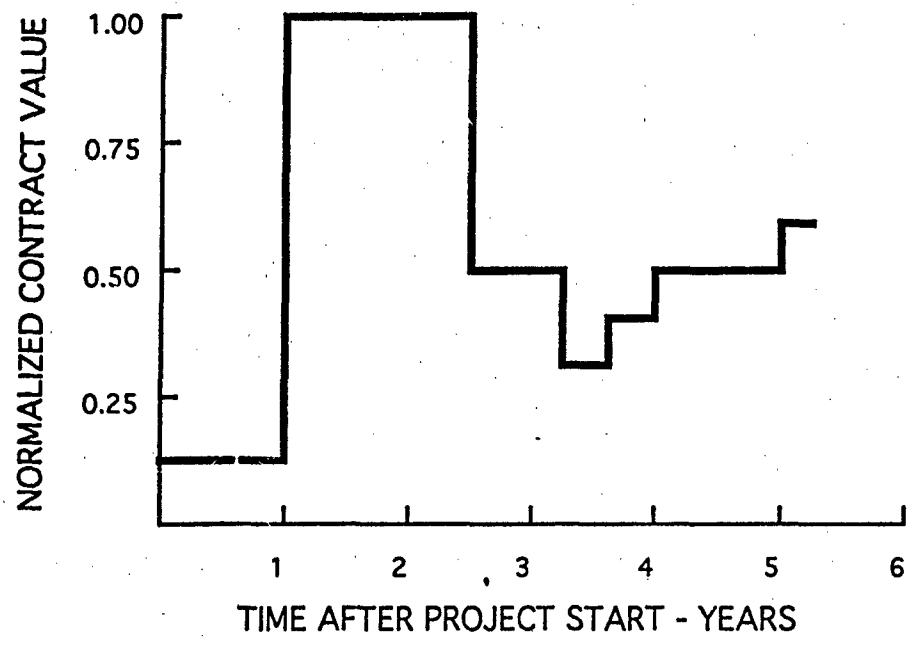


Figure 1 - Normalized Budget History of a Project

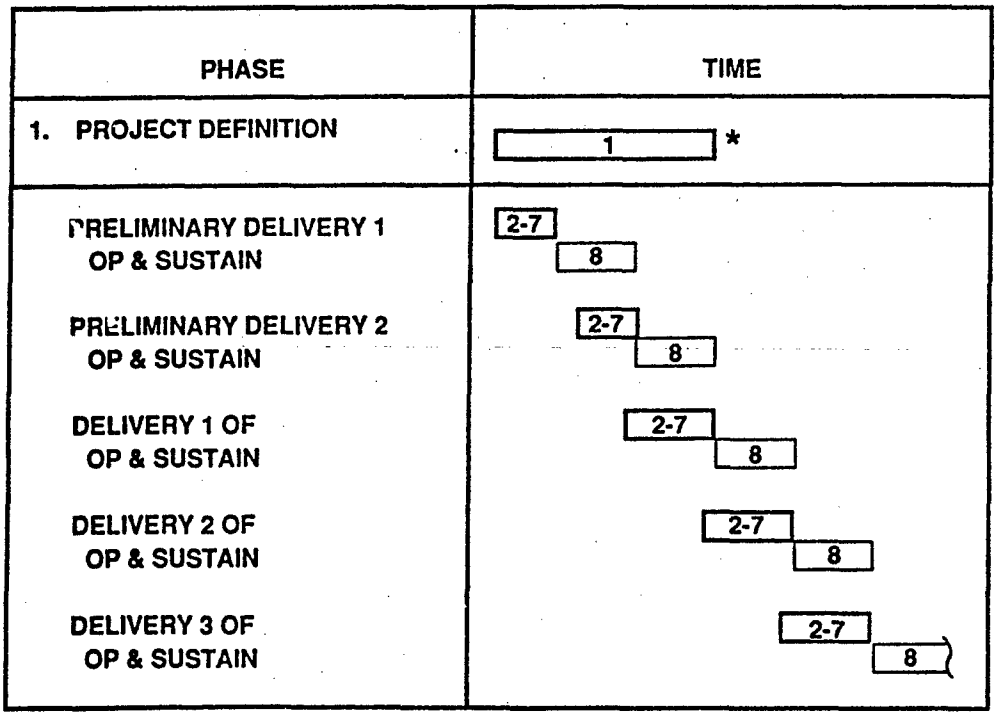


Figure 2 - Generic Project Life Cycle Using Rapid Development Methodology

DOCUMENT CATEGORIES	EARLY DELIV	MID DELIV	LATE DELIV
REQUIREMENTS SPECIFICATION	FUNCTIONAL REQUIREMENTS	SECURITY & PERFORMANCE	-ILITIES
DESIGN DOCUMENTS	REVIEW MTLS & PKG SPECS	DETAIL DESIGN	DETAIL DESIGN
INTERFACE CONTROL DOCUMENTS	FUNCTIONAL REQUIREMENTS	PERFORMANCE	-ILITIES
AS BUILT SPECIFICATIONS	DRAWINGS & LISTINGS	DRAWINGS & LISTINGS	DRAWINGS & LISTINGS
USER MANUALS	BASIC FUNCTIONS	ADVANCED FEATURES	EXCEPTION HANDLING
TEST DOCUMENTS	PLANS & OPERATIONS	PROCEDURES & FUNCTIONS	ACCEPT-ANCE
LOGISTICS DOCUMENTS	INITIAL PLANS	PLANS	MANUALS

Figure 3 - Progression of Documentation Formality

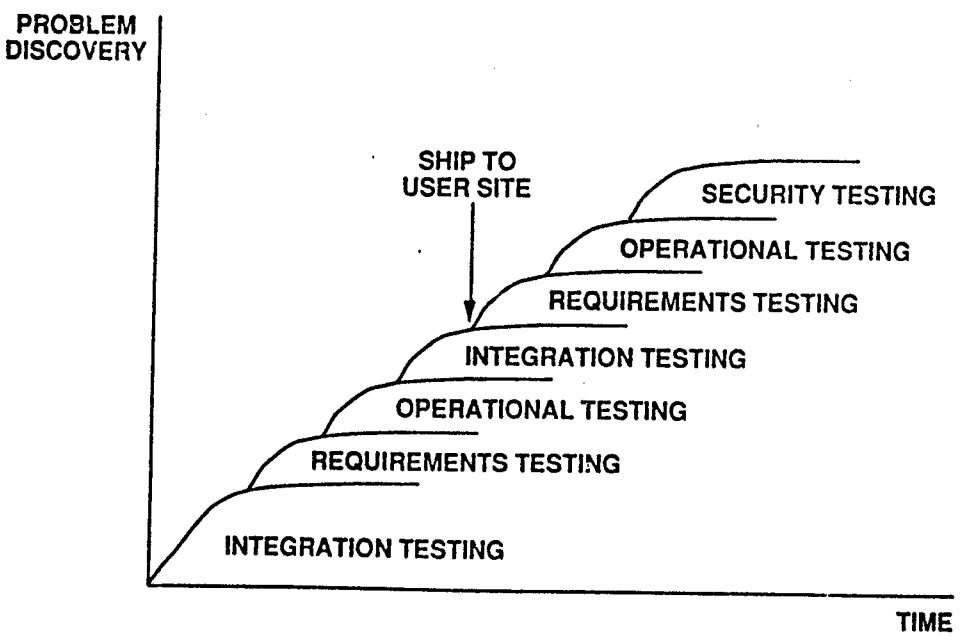


Figure 4 - The Effect of Testing in Multiple Environments

A FARMER'S GUIDE TO OOA: HARVESTING REQUIREMENTS

Jeffery D. Boyken
Coleman Research Corporation, Huntsville, Alabama

Brian K. Mitchell
Coleman Research Corporation, Huntsville, Alabama

Michael J. O'Connor
Coleman Research Corporation, Huntsville, Alabama

Abstract

Traditionally there has been a disconnect between software requirements and software design in large defense systems. The problem begins with the major product of the requirements phase, the Software Requirements Specification (SRS). Typically this document is intended for management and the customer not the design engineer. This paper describes a process to bridge the gap between the requirements phase and the design phase. This process is called a requirements harvest. The requirements harvest is a formal process for handing over requirements form the requirments engineers to the design engineers.

Introduction

Traditionally there has been a disconnect between software requirements and software design for large defense system using DoD-STD-2167A. There are several causes for this problem. The problem begins with the major product of the requirements phase, the Software Requirements Specification (SRS). This document is intended for management and the customer not the design engineer. As more complex systems are build, it becomes increasingly difficult to hand over large sets of requirements to the design engineer.

Many assumptions are made as requirments are defined. In general, these assumptions are poorly documented. There are of course other causes for breakage between the requirements phase and the design phase of a large software development project. This problem has been recognized by the Software Engineering Institute (SEI) in their Contractor Maturity Model (CMM) [HUM87].

The requirements harvest process was defined to resolve the requirements hand-over problem. The term 'requirements harvest' is used because the process of performing requirements analysis is like planting a crop and tending it. While you may have grown a bountiful crop (complete requirements), if you do not effectively harvest the crop it will rot in the field. Frequently a detailed requirements crop is produced, but the design engineers fail to utilize the yield. Instead, they hurry to the grocery store and pickup what they can find leaving the requirements to ruin.

The requirements harvest concept will work with most requirements analysis and design methods. However, the process works particularly weil with an object oriented approach. The authors used the Coad-Yourdon method for object

oriented analysis (OOA) [COA91] and a modified Buhr and Booch method for object oriented design (OOD) [BOO91]. The system was implemented in Ada which does not support full object oriented programming (OOP). This paper describes the harvest process in the context of the aforementioned methods.

First, this paper defines the requirements hand-over problem as it relates to large DoD programs. Next the harvest process is defined in detail, followed by the authors experience in applying the process. Finally, the benefits are presented.

Problem

Many problems face today's software development teams. Each time a software development team solves one set of problems, another set is created. The following paragraphs summarize problems encountered as the software engineering process matures.

Two major problems have prevented design engineers from designing a system that satisfies the customers expectations. First, requirements are traditionally incomplete and incorrect. Second, when the requirements are detailed and complex, as they are with large systems, the development process does not ensure that the design engineers completely understand the requirements. The combination of these two problems can be devastating. When a large number of errors exists in a large software requirements specification, it is nearly impossible to utilize it. Because of these problems, design engineers are forced to develop their own requirements. The result is a design that does not satisfy anyone's expectations, except possibly the design engineers.

A well defined software requirements analysis process assists the requirements engineer in defining requirements that have minimal errors. As a result the detail of the specifications will increase. Therefore, in solving the first problem (excessive requirements) the second problem (passing requirements on to the design engineers) is magnified.

If a good specification is developed for a large complex system, some errors will still exist after the software requirements review. If the design engineers do not have a thorough understanding of the specification they cannot identify requirements errors as they progress into design. As a result, design errors will be introduced because of misinterpreted requirements.

Once the software development team has developed complete requirements, more problems are encountered. In a waterfall approach, software requirements are developed early in the program and never updated. After the initiation of the design phase, little effort is expended to keep the requirements up to date. As a result, the requirements almost never represent what is designed.

To summarize the problems, improving requirements specification increases requirements detail and complexity. These detailed and complex requirements are difficult to pass on the design engineers such that they can satisfy them and identify remaining errors. In addition, these requirements are not maintained through the design phase. Recognizing that there is no silver bullet for software engineering, the requirements harvest process resolves several of these significant problems.

Process

The requirements harvest process is a formal three step process. As a formal process the harvest should be documented by forms (similar to walk-through forms). These forms include checklists that aid the reviewers in reviewing the pertinent data. The steps of the harvest are very similar to

structured walk troughs done in the coding phase. Weber's *Key Practices of the Capability Maturity Model* describes many of the processes that need to be performed in software development [WEB91] of which the requirements harvest is just one.

Step 1

The software specification review (SSR) initiates hand-off of the software requirements from requirements engineers to design engineers. At the SSR the design engineers get a thorough overview of all the requirements for each CSCI (Computer Software Configuration Item). The SSR provides a general overview of the requirements and is not intended to discuss each requirement in detail. For a large system, a general requirements overview (as called-out by MIL-STD-1521B) requires several days to complete.

At the completion of the SSR a requirements harvest is initiated to ensure that the design engineers are intimately knowledgeable of the requirements they are designing toward. The requirements harvest is initiated by the lead design engineer. The lead design engineer assigns the requirements objects, from the object-oriented requirements model, to each of the design engineers. Requirements are harvested on an object-by-object basis. After assigning objects, the lead requirements engineer schedules a series of requirements reviews, or harvests, with the design engineers. Each harvest is supported by:

1) the lead requirements engineer,
2) the requirements engineer responsible for specification of the requirements object,
3) the lead design engineer,
4) the design engineer responsible for designing-to the requirements object.

These harvests allow the requirements engineers to review in detail the requirements that the design engineer is responsible for satisfying.

The basic premise of the initial harvest review is for the requirements engineer to explain the requirements to the design engineer. He will have to resolve any anomalies (ambiguities, errors, and inconsistencies) that exist. Each anomaly is identified through the initial harvest. Additional specification may be required to permanently resolve the anonolies. The requirements engineer leads the initial harvest. In the harvest, the requirements object is discussed in detail as well as its relationships to other requirements objects. The initial harvest includes a detailed walk-through of the following specifications:

- attributes [COA91]
  - for each attribute: description, purpose, accuracy, range, precision, traceability
  - class unique, object unique, generalizations, and specializations
- services [COA91]
  - for each service: description, purpose, inputs, outputs, timing, traceability
  - class unique, object unique, generalizations, and specializations
  - implied services (create, delete, get, set)
- relationships to other objects
  - for each instance connection [COA91]: attributes required, purpose for the requirement
  - for each message connection [COA91]: services supplied, services requested
  - for each part and whole relationship[COA91]: relationships to other objects

During the initial harvest, the requirements engineer is responsible for completing software change requests

(SCRs) for all requirements errors identified in the review. The initial harvest ensures all requirements have been reviewed in detail for design considerations. The lead requirements engineer reviews all the SCRs generated and corrects all approved SCRs. When all the SCRs are closed, a new object oriented requirements model is released. After the new model is released, the lead requirements engineer schedules a second series of reviews to discuss changes.

Step 2

Once the design engineers have completed their preliminary design, Ada Buhr diagram specifications and PDL package specifications, the design engineer walks the requirements engineer through the design specifications. This second harvest identifies additional errors in the requirements and ensures that the requirements are satisfied by the design. Again, the requirements engineer is responsible for completing all SCRs against the requirements. Once the SCRs are closed the requirements engineer discusses the changes with the design engineers. The preliminary design review (PDR) is held following the closure of all SCRs related to the preliminary design level harvest.

Step 3

The requirements harvest process is completed prior to the critical design review (CDR). When a design engineer has completed his Buhr diagram bodies and Ada PDL bodies, he meets with the requirements engineer. The design engineer walks the requirements engineer through the design specification to identify any requirements errors and to ensure that the requirements are satisfied. Again, the requirements engineer is responsible for completing all SCRs and coordinating their resolution with the design engineer. The CDR is held following the closure of all SCRs related to the detail design level harvest.

Experience

The authors have used the requirements harvest on a large distributed real-time defense system. After the SRR, the lead design engineer created a design architecture based on the requirements model. Individual design engineers were then assigned to each requirements object. The harvest process was initiated following the assignments. The harvest was the first activity of the preliminary design process.

The harvest requires a form to document and guide the processes. The inclusion of this form is the result of the first use of the process. When the harvest was first conducted only informal notes where keep by the lead requirements engineer and lead design engineer. The informal notes did not provide adequate visibility into the process.

This project marked the first time the software development team performed object oriented requirements analysis. While all of the development team received OOA training, only the requirements engineers were experts in the method and notation. The requirements harvest enabled the requirements engineers to explain the OOA notation in detail with the design engineers. Thus the harvest eased the paradigm shift.

Due to program constraints, the requirements analysis process was not allocated adequate resources. Additionally, systems engineering did not produce a complete set of systems requirements in time to "seed" the software requirements process. As a result, the software requirements were incomplete. These problems were identified before the requirements harvest. However, the harvest provided metric data, SCRs, to make this problem more evident to management.

Management supported the harvest process, but did not commit adequate resources. The requirements model contained approximately 100 objects. The design team consisted of 7 engineers including the design lead. Management expected the harvest to be completed in two to three days. Initial estimates by the software engineers indicated that it would require 10 days. The harvest actually took 3 weeks to complete. with the average object requiring about one hour of discussion.

Over 100 SCRs were generated during the harvest. This number was much larger than expected. The large number of SCRs was attributed to several factors.

1. Incomplete system requirements
2. Inadequate resources to develop the requirements
3. Limited experience with OOA

We anticipate that a program with better system requirements and more resources spent on analysis would not generate as many SCRs. On this particular program, the errors were found before the design phase. This significantly reduced the cost of resolving the anomalies. Without the harvest, most of these errors would not have been found until later phases of development, thus resulting in increased development cost and schedule overruns.

Conclusion

Many DoD systems being developed today are very large and complex. For development methodologies to be successful on these programs, they must produce detailed requirements specifications. These detailed requirements specifications must be managed carefully to ensure that:

1) design engineers understand the requirements specification,
2) the design process identifies requirements errors,
3) the requirements and design remain consistent.

The requirements harvest process has been defined to manage these requirements issues. The requirements harvest is successful because it formally ensures:

1) requirements are understood by the design engineers before preliminary design begins,
2) requirements include the design engineers perspective,
3) requirements are incrementally verified through the design process.

For the farmer of the future to be successful he must improve his processes. He must utilize new technologies to produce larger crops with less resources and with increased yields. An efficient requirements harvest is essential to realizing increased yields.

References

[BOO91] Booch, Grady, Object Oriented Design With Applications, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991.

[COA91] Coad, Peter and Edward Yourdon, *Object-Oriented Analysis, Second Edition,* Yourdon Press, Englewood Cliffs, New Jersey, 1991.

[HUM87] Humphrey, Watts S., W. L. Sweet, *A Method for Assessing the Software Engineering Capability of Contractors*, Software Engineering Institute, Pittsburgh, Pennsylvania, 1987.

[WEB91] Weber, Charles V., Mark C. Paulk, Cynthia J. Wise, James V. Withey, *Key Practices of the Capability Maturity Model*, Software Engineering Institute, Pittsburgh, Pennsylvania, 1991.

Biographies

JEFFERY D. BOYKEN is currently employed as a Software Engineer for Coleman Research Corporation in Huntsville, Alabama. He holds a Bachelors of Science in Engineering Physics from Murray State University. He

specializes in Object Oriented software requirements analysis for large distributed real-time systems. Mr. Boyken drives a 1987 Buick Grand National.

BRIAN K. MITCHELL is currently employed as a Software Engineer for Coleman Research Corporation in Huntsville, Alabama. He holds a Bachelors of Science in Computer Science from Murray State University and a Masters of Science in Computer Science from The University of Alabama in Huntsville. He specializes in system and software requirements analysis for large distributed real-time systems. Mr. Mitchell drives a 1966 Ford Thunderbird.

MICHAEL J. O'Connor is currently employed as a Software Engineer for Coleman Research Corporation in Huntsville, Alabama. He holds a Bachelors of Computer Engineering from Auburn University and a Masters of Science in Computer Science from The University of Alabama in Huntsville. He specializes in the development of large distributed real-time systems in Ada for defense applications. Mr. O'Connor drives a 1975 Oldsmobile Delta 88 Royal convertible.

Ada PERFORMANCE ISSUES IN REAL-TIME TRANSPUTER ENVIRONMENTS

Richard M. Plishka
Computing Sciences Department
University of Scranton
Scranton, PA 18510
717-941-6111
email: plishka@jaguar.uofs.edu

ABSTRACT

This article relates the experiences of a project undertaken at the Chemical Research, Development and Engineering Center of the U.S. Army. The objective of the project was to determine the feasibility of a real-time Ada implementation on a transputer-based embedded system. Benchmarks were performed in Ada and OCCAM on the 80x86 and T800 platforms. This report contains timing comparisons of the Whetstone and PIWG benchmarks on these platforms.

INTRODUCTION

The Detection Directorate of the Chemical Research, Development and Engineering Center (CRDEC) is developing an embedded system which utilizes the INMOS T800 Transputer. Although there are several programming languages available for systems development on the Transputer, one of the most widely used is OCCAM. OCCAM's popularity can be attributed to the fact that it is a high level language designed to express parallel algorithms and their implementation on a network of processing components. In addition, the Transputer may be considered an OCCAM machine; OCCAM provides the efficiency equivalent to that of programming a conventional computer at the assembly language level [INMOS 88]. However, given the Congressional Ada Mandate (Public Law 101-511 - Sec. 8092, and Public Law 102-172, Sec. 8073), Ada has been designated the systems development language of choice for Department of Defense software projects. The objective of this project was to evaluate the Alsys_037 Ada compiler for the Transputer, currently the only commercially available Ada compiler for the Transputer, in order to determine the feasibility of implementing the required software in Ada.

The general approach that was taken for this project was to run a series of software benchmark tests conforming to *figure 1.*

	80x86 - DOS 5.0		TRANSPUTER	
BENCHMARK	Alsys Ada	Meridian Ada	Alsys Ada	Occam
Whetstone	X	X	X	X
PIWG	X	X	X	
Hartstone	X	X	X	

figure 1 - Test Plan

The Whetstone benchmark program [Curnow 76] was developed to compare processing power for scientific applications. The program goes beyond measuring pure floating point performance ('flops') by including features found in 'typical' scientific applications such as: conditional jumps, array indexing, integer arithmetic, procedure calls, and evaluation of elementary functions. The PIWG test suite [Pollack 90, Roy 90] contains a series of experiments that assist in the evaluation of processor performance, clock resolution and compilation efficiency. Hartstone [Weiderman 89] is a benchmarking tool for evaluating hard real-time performance.

The initial plan was to test these software benchmark systems across the Intel 80x86 and INMOS T800 platforms using the Alsys Ada compiler for 32-bit DOS, the Alsys Ada compiler for the Transputer, and the Meridian Ada compiler for 32-bit DOS. The quantitative results from these tests would then be used as the basis for conclusions and recommendations.

TECHNICAL DISCUSSION

HARDWARE/SOFTWARE

The hardware system which was used to develop and test the benchmarks consisted of:

Host system - Gateway 2000: 80486DX/33MHz, EISA, 8MB RAM
Transputer - CSA Transputer board for PC: T800/20MHz, 4MB RAM

Once the DOS executables were generated, timings were measured on the 386/20MHz, 386/33MHz, and 486/33MHz systems. The 386 systems were equipped with 80387 math coprocessors.

The systems software used to support the project consisted of:

DOS 5.0
Alsys Ada - version 5.1 - 32 bit DOS compiler
Alsys Ada_037 - version 5.4.2 - Transputer compiler
Meridian Ada - version 4.1.1 - 32 bit DOS compiler
INMOS Occam Toolset - version D7205

Early in the project, the Meridian compiler was abandoned due to the volume of compile-time and run-time errors encountered with code which was successfully tested in the Alsys environment. Time did not permit debugging and rewriting a large volume of code. Therefore, the revised test plan matrix conforms to *figure 2*.

	Alsys Ada - 32 bit DOS			Transputer	
BENCHMARK	80386/20MHz	80386/33MHz	80486/33MHz	Alsys Ada	Occam
Whetstone	X	X	X	X	X
PIWG	X	X	X	X	
Hartstone	X	X	X	X	

figure 2 - Test Plan - revised

ACTIVITIES

WHETSTONE The Whetstone benchmark has been considered somewhat a standard benchmark for a number of years. Unlike the Drystone benchmark, Whetstone is intended to simulate 'typical' scientific applications through its utilization of a variety of routines. It does, however, fall short in several categories of contemporary scientific calculations. In particular, it uses small arrays, no multi-dimensional arrays are employed, it is dependent on the speed of floating point operations, and the number of elementary function evaluations is probably atypical of current programming models [INMOS 91, p259]. Despite these observations, it still provides a legitimate baseline for the evaluation required in this project. Whetstone was

successfully coded and tested in both Ada and Occam, and run on the 80x86 and T800 platforms.

PIWG The Performance Issues Working Group of the ACM has made available a series of Ada benchmarks which can be used in the evaluation of Ada compilers across a range of hardware platforms. The test suite assists in the evaluation of execution time and compilation time. It was determined for this project that the compilation time tests were of little value at this stage in the project; therefore, emphasis was placed on the evaluation of execution timings. In particular, there are four test areas that made up the critical area of performance testing. They are:

1. Clock Resolution (A000090) This test illustrates CPU clock resolution available to Ada.
2. DELAY Resolution (Y000001) Measures the resolution of the DELAY feature of Ada.
3. Procedure Call Overhead (P000005/6/7) Measures procedure call overhead time in Ada.
4. Hennesy Tests (A000094A-K)Series of tests which measure performance in a variety of areas including: recursion, integer and real matrix multiplication, and sorting (data movement).

These tests were successfully performed in Ada on both the 80x86 and T800 platforms.

HARTSTONE The Hartstone benchmark is a set of timing requirements for testing a system's ability to handle hard real-time applications [Weiderman 89]. The complete Hartstone benchmark consists of five categories of tests: PH Series, PN Series, AH Series, SH Series, and SA Series [Weiderman 89, p5]. The only test successfully implemented in Ada to date is the PH Series. This test provides feedback for a set of tasks which are periodic and harmonic.

The PH Series was successfully tested on the T800 platform in Ada; however, the 80x86 DOS tests failed to provide reliable results. Best estimation is that as the period in milliseconds began to approach the clock resolution available through the Ada/DOS environment, the system would "hang"; apparently attributable to DOS. Therefore, the quality of comparable 80x86/T800 results was compromised. Because of deadline constraints, it was determined that the Hartstone benchmarks could not be successfully implemented and therefore omitted from this report.

PROCEDURES

All benchmark development, testing and implementation was performed on the hardware and software previously noted. Once the executables were generated, testing was conducted according to *figure 2* . All compilations and binding/linking optimization options are contained in **APPENDIX A**.

RESULTS

Test results are contained in figures 3 - 5. A key point worth noting is that the Ada/Transputer environment is, in effect, a runtime environment. That is, the execution of the Ada generated code is supervised by the ISERVER. This runtime environment does not permit the Ada code access to the high priority one microsecond clock resolution available on the T800. The OCCAM environment, however, does permit OCCAM code one microsecond access. Therefore, perhaps one of the most interesting charts is the WHETSTONE comparison. The other charts, however, do provide valuable information on the comparison of Ada executables on 20MHz and 33MHz processors utilizing different architectures.

CONCLUSIONS

The CBMS under development by CRDEC requires one microprocessor feature that is not supported by the 80x86 line of processors; that is, the requirement to have < 3 microsecond resolution. The dilemma highlighted by this study concludes that the current Ada environments available fall short of providing this feature on the T800, even though it can be supported via OCCAM.

A variety of options exist in the pursuit of a solution to this problem. Perhaps the most interesting would be that of developing the CBMS software support system using both Ada and OCCAM. This option may satisfy both the timing constraints of the CBMS project as well as the Congressional Ada Mandate. One item missing in permitting this recommendation is evidence of the real-time timing requirements of OCCAM. Comparative data, such as that provided in this report, illustrating OCCAM's statistics in similar PIWG and Hartstone

implementations would be helpful. Such data could provide valuable insight as to whether or not OCCAM may be a viable alternative or supplement to the Ada development environment.

ACKNOWLEDGEMENTS

This work was supported by the Chemical Research, Development and Engineering Center under the auspices of the U.S. Army Research Office Scientific Services Program administered by Battelle (Delivery Order 251, Contract No. DAAL03-91-C-0034).

REFERENCES

[Borger 89] Borger, M., Klein, M., Veltre, R.
Real-Time Software Engineering in Ada: Observations and Guidelines
Technical Report CMU/SEI-89-TR-22, DTIC: ADA219020,
Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, September, 1989

[Curnow 76] Curnow, H.J., and Wichmann, B.A.
A Synthetic Benchmark
Computer Journal 19 (1), 43-49, January, 1976

[Donohoe 90] Donohoe, P., Shapiro, R., Weiderman, N.
Hartstone Benchmark User's Guide, Version 1.0
Technical Report CMU/SEI-90-UG-1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, March, 1990

[INMOS 88] INMOS Limited
OCCAM 2 Reference Manual
Prentice Hall, New York, 1988

[INMOS 89] INMOS Limited
Transputer Applications Notebook - Systems and Performance
Redwood Press Ltd., Melksham, 1989

[INMOS 91] INMOS Limited
OCCAM 2 Toolset
SGS-Thomson Microelectronics Inc., Colorado Springs, 1991

[LRM 83] United States Department of Defense
Reference Manual for the Ada Programming Language
American National standards Institute, New York, 1983

[Pollack 90] Pollack, R.H., and Campbell, D.J.
Clock Resolution and the PIWG Benchmark Suite
Ada Letters - Special Edition on Ada Performance Issues X(3), 91-97, 1990

[Roy 90] Roy,D., and Gupta, L.
PIWG Analysis Methodology
Ada Letters - Special Edition on Ada Performance Issues X(3), 217-229, 1990

[Weiderman 89] Weiderman, Nelson
Hartstone: Synthetic Benchmark Requirements for Hard Real-Time Applications
Technical Report CMU/SEI-89-TR-23, DTIC: ADA219326,
Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, June, 1989

Richard M. Plishka is an Associate Professor in the Computing Sciences Department. His research interests include Ada development environments and real-time parallel systems. He is a member of the ACM and SIGAda.

CLOCK RESOLUTION (A000090)

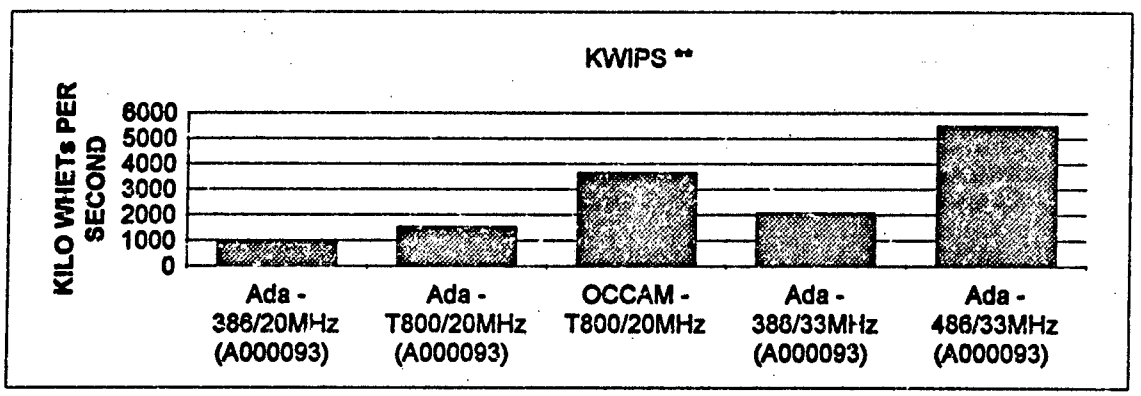
Alsys Ada

System	Time (seconds)
Transputer (T800-20/20MHz)	0.000061035156250
30386/20MHz (DOS)	0.000054492187500
80386/33MHz (DOS)	0.000054492187500
80486/33MHz (DOS)	0.000054492187500

WHETSTONES

TARGET	KWIPS **	RunTime File Size (in bytes)	File Type
Ada - 386/20MHz (A000093)	1010	126944	.EXE
Ada - T800/20MHz (A000093)	1541	55890	.BTL
OCCAM - T800/20MHz	3655	9646	.BTL
Ada - 386/33MHz (A000093)	2075	126944	.EXE
Ada - 486/33MHz (A000093)	5489	126944	.EXE

** Kilo Whetstone Instructions Per Second



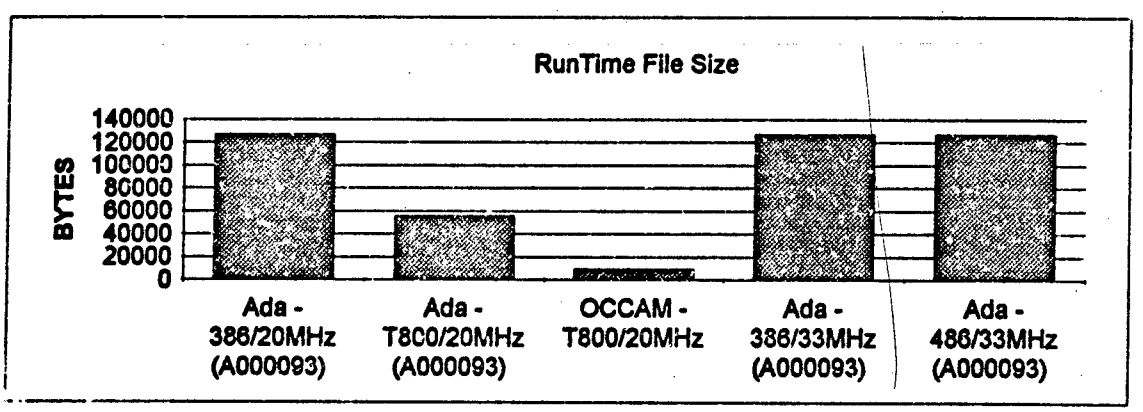
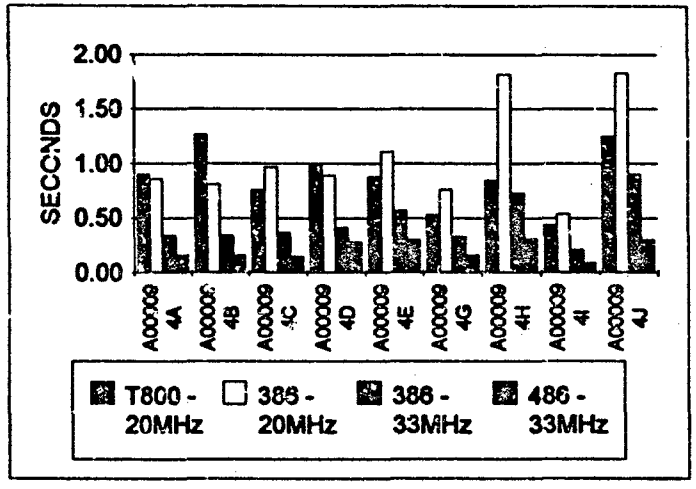


figure 3 - Clock Resolution/Whetstone Results

Hennesy Tests - Ada Results

Test Description	TEST ID	T800 - 20MHz	386 - 20MHz	386 - 33MHz	486 - 33MHz
Permutations - 43,300 - highly recursive	A000094A	0.90	0.86	0.34	0.16
Towers of Hanoi - highly recursive	A000094B	1.27	0.81	0.34	0.16
Eight Queens - solved 50 times	A000094C	0.76	0.97	0.37	0.15
Integer Matrix Multiply - 40X40	A000094D	0.98	0.89	0.41	0.28
Real Matrix Multiply - 40X40	A000094E	0.88	1.11	0.57	0.30
Quicksort - 5K random integers	A000094G	0.53	0.76	0.33	0.16
Bubblesort - 5K random integers	A000094H	0.65	1.82	0.73	0.31
Tree Insertion Sort - 5K random integers	A000094I	0.44	0.54	0.21	0.09
FFT (256 point complex) - performed 20 times	A000094J	1.25	1.83	0.90	0.30
Puzzle Problem - CPU intensive	A000094F	5.27	4.57	2.04	0.91
Ackermann's Function - performed 10 times	A000094K	10.85	11.41	4.44	1.93



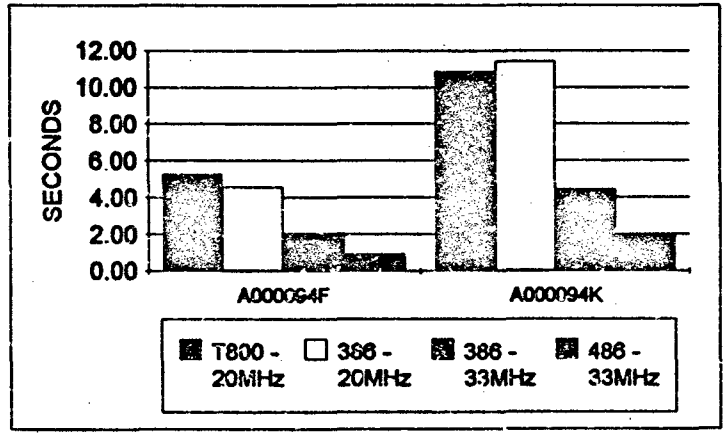


figure 4 - Hennesy Test Results (PIWG Test Suite)

DELAY RESOLUTION (Y000001)

Commanded	Transputer (T800-20/20MHz)	386/20MHz (DOS)	386/33MHz (DOS)	486/33MHz (DOS)
0.00097	0.00115	0.00171	0.00171	0.00171
0.00195	0.00213	0.00256	0.00256	0.00256
0.00390	0.00415	0.00427	0.00427	0.00427
0.00781	0.00805	0.00854	0.00854	0.00854
0.01562	0.01599	0.01630	0.01630	0.01630
0.03125	0.03161	0.03174	0.03174	0.03174
0.06250	0.06280	0.06262	0.06262	0.06262
0.12500	0.12530	0.12525	0.12524	0.12524
0.25000	0.25024	0.25055	0.25055	0.25055
0.50000	0.50036	0.50031	0.50031	0.50031
1.00000	1.00030			1.00067
2.00000	2.00024			2.00049

PROCEDURE CALL OVERHEAD

Alsys Ada - time in microseconds

Test	Transputer (T800-20/20MHz)	386/20MHz (DOS)	386/33MHz (DOS)	486/33MHz (DOS)
P000005	5.27	4.04	1.63	0.84
P000006	4.66	4.16	1.72	0.80
P000007	5.03	4.51	1.79	0.98

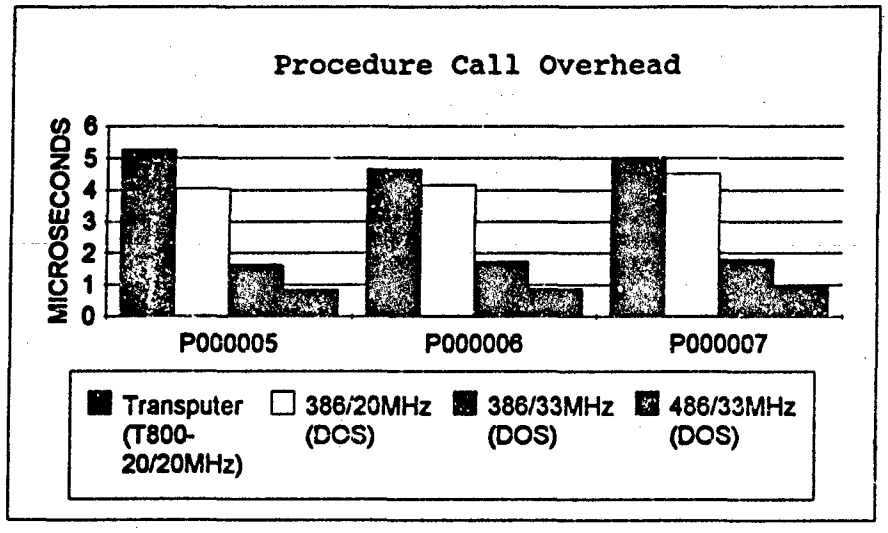


figure 5 - Delay Resolution/Procedure Call Overhead

APPENDIX A
OPTIMIZATION

All code was compiled and linked taking advantage of optimization features provided by each specific development environment.

Ada - DOS

Compiler - Alsys Ada for 32-bit DOS - version 5.1

Compile options - IMPROVE=(CALLS = INLINED
REDUCTION = EXTENSIVE
EXPRESSIONS = EXTENSIVE)

CALLS = INLINED: Call will be inlined for subprograms that aren't directly or indirectly recursive in response to INLINE pragma.

REDUCTION = EXTENSIVE: Performs analysis of intermediate program representation to eliminate numerous run-time checks and removal of dead code.

EXPRESSIONS = EXTENSIVE: Performs common subexpression elimination and additional register optimization.

Bind options - TIMER = FAST

TIMER = FAST: High resolution timer used for the implementation of the DELAY statement.

Ada - Transputer

Compiler - Alsys Ada for the Transputer - version 5.4.2

Compile options - IMPROVE=(INLINE = PRAGMA
REDUCTION = EXTENSIVE
EXPRESSIONS = EXTENSIVE)

INLINE = PRAGMA: Same as CALLS = INLINED above.

REDUCTION = EXTENSIVE: Same as REDUCTION = EXTENSIVE above.

EXPRESSIONS = EXTENSIVE: Same as EXPRESSIONS = EXTENSIVE above.

Bind options - FAST_MAIN = YES, FAST_TASK = YES

FAST_MAIN = YES: Attempt to allocate the primary stack of the main program in a low-addressed area which could be mapped to the internal on-chip memory of the Transputer.

FAST_TASK = YES: Attempt to allocate the primary stack of the task in a low-addressed area which could be mapped to the internal on-chip memory of the Transputer.

OCCAM

Compiler - INMOS OCCAM Toolset - version D7205

Compiler options - /a /t8 /h

/a: Prevents compiler from performing alias checking, and prevents usage checking.

/t8: Compile for T800 processor.

/h: Produces code in HALT mode.

Linker options - /t8 /h

/t8: Specifies T800 as target processor.

/h: Generates a linked unit in HALT mode.

Code Collector options: /t

/t: Creates a bootable file for a single transputer.

Host file server: /se

/se: Terminates the server if the Transputer error flag is set.

Defense Software Repository System Panel

Moderator: Joanne Piper, DISA/CIM

Panelists: Marrea Riggs, Army
Patti Hicks, Defense Logistics Agency
Rob Rutherford, Air Force, Standard Systems Center
Jim Wheeler, Navy

Ada in Undergraduate Computing Education: Experience & Lessons Learned

Moderator: John Beidler, University of Scranton

Panelists: Mike Feldman, George Washington University
Nick DeLillo, Manhattan College
Jim Smith, Leymoye College
John W. McCormick, State University of New York

Programming In the Large

Moderator: Dr. Donald Mullikin, FAA

Panelists:

Reuse Interoperability Group (RIG)

Moderators: Jim Moore, IBM
Dave Dikel, Applied Expertise, Inc.

Panelists: Eric Beser, Westinghouse
Linda Braun, MountainNet
Pam Arya, General Research Corp.
David Dikel, Applied Expertise

TRANSITION TO ADA: A CASE STUDY

Urban LeJeune and Murray Kirch
Stockton State College
Pomona, New Jersey 08240

SUMMARY

This paper describes, in case study format, the pedagogical change to Ada from Pascal at Stockton State College. The transition was started in the Fall 1988 semester. During that semester Ada was introduced into the Operating Systems and Programming Language Structures courses. The metamorphosis was complete in the Fall 1991 semester with the adoption of Ada in our Programming and Problem Solving I course, which is based on the ACM CS1 guidelines.

The experience has proven to be pedagogically sound and enthusiastically supported by both faculty and students. A key to the successful transition was the initial introduction of Ada at the senior level and subsequently incorporating the use of the language progressively lower in the curriculum. The philosophy was based upon the premise that you do not have to teach upper level students how to program and, additionally, upper level students, after being exposed to Ada, would become formal and informal tutors and laboratory assistants.

GENERAL BACKGROUND

Stockton State College is a moderately sized liberal arts college located in Pomona, New Jersey which is about twenty minutes to the Atlantic City boardwalk. The school has slightly over 5,000 full time equivalent students and offers degrees in a variety of liberal arts and professional majors. The Computer and Information Sciences (INFO) program is domiciled in the Professional Studies Division. Professional Studies also includes Business Studies and a variety of health related programs. The INFO program supports approximately 125 majors and has eight full-time and one half-time faculty members.

The INFO program offers both BA and BS degrees. The curriculum is based upon a common core of courses which is required for all majors. BS candidates chose between four major tracks while BA candidates tailor their programs to satisfy their individual career goals including a broader liberal arts component.

The common core of courses required of all INFO majors are:

INFO 1206	Statistics I
INFO 2101	Programming and Problem Solving I
INFO 2102	Programming and Problem Solving II
INFO 2210	Systems Analysis and Design
INFO 2222	Fundamentals of Information Systems
MATH 2225	Discrete Mathematics I
MATH 2215	Calculus I

The four concentration tracks for BS candidates are Computer Science, Information Systems, Management Information Systems, and Computer Education. All four tracks require 51 credits of applicable concentration and cognate coursework in addition to the common core. BA candidates are required to complete 35 credits of coursework in computer and cognate courses above the common core requirement.

ADA BACKGROUND

In the early eighties Stockton embraced the educational concept of emphasizing a single programming language for instructional purposes as opposed to a sampling of languages that was prevalent at the time. The language of choice was Pascal. The only time other languages were taught, and continue to be taught, is when they have application features not included in the primary language. File processing using COBOL, numerical methods using FORTRAN, and artificial intelligence using LISP and Prolog are examples.

As the decade progressed, the Stockton computer science curriculum placed increasingly greater emphasis on emerging software engineering concepts. It became painfully clear that standard Pascal lacked features that would provide strong support for major software engineering principles. Consequently, faculty decided to examine other languages to determine one most appropriate for the INFO program.

One of the primary goals in the instructional language selection process was the capacity of a language to be broadly included in the curriculum. Wherever possible, within the limits of a liberal arts college curricular requirements, the Stockton program incorporates the ACM curriculum guidelines.

After an extensive evaluation of many languages, Ada was selected as the language of instruction. The support for software engineering principles plus an exceptionally high level of standardization made Ada the clear choice. Additional factors were the emergence of validated Ada compilers for IBM PC compatible platforms and Stockton's close proximity to the Federal Aviation Agency Technical Center in Pomona, New Jersey. Stockton has historically placed many cooperative education students at the Tech Center.

ADA EVOLUTION

Murray Kirch, the senior faculty member in the department, was instrumental in the selection process and the subsequent faculty training. It is axiomatic that there must be a strong and dedicated leader if a project of this size is to be carried to fruition.

Before implementing Ada in any course, substantial groundwork and preparation is required. If the program is to succeed, there must be strong faculty preparation. The Stockton transitional process commenced with Murray Kirch attending an intensive 4-week faculty seminar at Tuskegee University. Later Murray conducted a one week Ada workshop for faculty members on the Stockton campus. Non-computer science faculty members were especially encouraged to attend the workshop that was supported by the institution.

It was decided to start the Ada transition process by initially employing the language in upper level courses. This would enable faculty to gain experience in using Ada with well-prepared students before attempting to introduce it in large, introductory level courses. This also produced a cadre of student assistants for lower

level courses to be taught in subsequent semesters.

The transitional goal was established to take place in a three year period. The goal for the first year was to introduce Ada into Junior/Senior level courses. The second year objective was to incorporate Ada in Sophomore level courses. The third year was the year that completed the process with the introduction of Ada into Freshman level courses which included the CS1 and CS2 courses.

During the fall semester of 1988 Ada was introduced to the Stockton curriculum with the offering of Operating Systems and Programming Language Structures. Ada's tasking ability made it an ideal programming language choice in the operating systems course. In the programming language course, Ada was both an object of study as well as the implementation language for a language translation project.

The spring semester of 1989 witnessed the incorporation of Ada as the language of choice for a software engineering course. Stockton also co-hosted the Seventh Annual National Conference on Ada Technology in Atlantic City during the semester. During the fall 1989 semester Ada was introduced into the sophomore level course in data structures.

During the fall of 1990 and the spring of 1991 Ada was the language used in the freshman level courses, Programming and Problem Solving II (CS2) and Programming and Problem Solving I (CS1). This completed the transition from Pascal to Ada.

The major Ada concepts introduced in the course sequence included the introduction of exception handling, packages, and attributes in the Programming and Problem Solving I (CS1) course. Generics, advanced exception handling, abstract data types and team oriented projects are embedded in our Programming and Problem Solving II (CS2) course. The Data Structures course enables students to gain experience with a more sophisticated use of generics and other Ada features. The Operating Systems course is a natural for the introduction of concurrency and tasking using problems such as the dining philosophers and the readers and writers.

Upper level courses featuring Ada features include Programming Language Structures (PLS) and Software Engineering. In the PLS course Ada is treated as an object of study and additionally is used as an implementation language to write a sophisticated project such as an interpreter for a Pascal type language. The Software Engineering may be conceptualized as a capstone course using large team projects encompassing both maintenance and startup projects. Because of the proximity of the Federal Aviation Agency Technical Center, and the fact that many students by this time have spend a semester co-op at the Tech Center, an air traffic control project is typically implemented. A maintenance project available from the Software Engineering Institute is frequently implemented as the maintenance component.

When Ada was first introduced the only compiler available to students was VAX Ada running on a DEC VAX cluster consisting of a VAX 6410 and a VAX 6310. In 1989 two Meridian compilers were made available to Stockton faculty through the Annual Conference on Ada Technology's Academic Outreach program. By 1990 a Novell network was outfitted with a Meridian Ada compiler, providing students and faculty with the option of using the PC/MS-DOS or VAX/VMS based product.

Meridian also made PC based compilers available to students at substantially reduced prices. Many Stockton students purchase Ada compilers to be used on their own computer systems.

OBSERVATIONS

Several lessons were learned during the process. Faculty development is an ongoing process with members attending frequent conferences and training seminars. Additionally, Stockton faculty have presented seminars and papers concerning technical and educational aspects of software engineering with Ada at regional, national, and international conferences.

Textbook selection presented an initial problem. The selection was, and is, limited when compared to the plethora of available Pascal books. The scarcity is especially noticeable at the introductory and intermediate levels. However, the quantity and quality of available Ada textbooks are dramatically increasing. A list of available textbooks appears in Feldman[1].

As a first programming language Ada does present some practical problems. Developmental environments are not as user-friendly as those available with Pascal. This deficiency is especially noticeable when compared with the exceptionally friendly front-end provided with Turbo Pascal. The increased time required to produce an executable program using Ada can be a source of student frustration.

Initial student programming frustration may be substantially overcome by the judicious use of supplied source code. A supplied package can hide many required implementation details from students until they are ready to comprehend the Ada language complexity.

CONCLUSIONS

The transition to Ada at Stockton has produced far more advantages than difficulties. Students are exposed to software engineering concepts starting with their first course. The use of Ada has facilitated more substantial student projects throughout the curriculum beginning with the introductory level course. A major side effect has been expanded student employment opportunities.

Student reaction to Ada has, in general, been enthusiastic. There is some initial reluctance from beginning students who have experience with Turbo Pascal.

Ada is a language that is designed to reduce life-cycle costs; this is partially accomplished by attempting to discover software errors as early in the life-cycle process as possible. As a consequence of the more extensive error checking performed by an Ada compiler, an Ada program written by a beginning student may be more difficult to successfully compile than its Pascal counterpart; however the Ada program is more likely to run successfully. However, this advantage is lost on some introductory programming students. The first programming course typically terminates with a demonstration of Ada's generic features. This characteristic, coupled with Ada's exception handling, tends to convert even the strongest Pascal proponents.

However, most of our students are eager to learn Ada. They know it is a more modern language and one for which there is strong local demand by prospective employers. Students also feel a justifiable sense of accomplishment as they learn to use Ada in their software projects.

A well prepared faculty, coupled with modest institutional support, resulted in a relatively painless transition to Ada. The "top down" approach of introducing Ada first in upper level courses and latter in the intermediate and introductory level courses worked well. (Feldman[1] provides several examples of academic institutions where Ada was introduced using a "bottom

up" approach). To paraphrase the old saw "you don't have to be a computer scientist to like Ada, try it you'll like it."

REFERENCE

1. Feldman, M. Ada experience in the undergraduate curriculum. *Commun. ACM 35*, 11 (Nov. 1992), 53-67.

About the Authors:

Urban A. LeJeune
Division of Professional Studies
Stockton State College
Pomona, New Jersey 08240
email: lejeune@pilot.njin.net
Phone (609)-652-4477

Dr. LeJeune is an associate professor of information and computer sciences at Stockton State College. He holds an Ed.D. in educational administration from Temple University, a M.S. in computer science from Monmouth College and a B.S.B.A. from Thomas Edison State College. He has twelve years of teaching experience in addition to over twenty years of computer and electronic industry experience.

Dr. LeJeune's research interests include computer science education, telecommunications as an educational tool, distant education, programming languages, program generators, language translators and the educational use of Internet. Ada activities have included presentations at several Ada specific conferences and publication on Ada and educationally related subjects.

Murray R. Kirch
Division of Professional Studies
Stockton State College
Pomona, NJ 08240
email: mrk@vax002.stockton.edu
Phone (609)-652-4353

Dr. Kirch is a professor of computer science and mathematics at Stockton State College. He received his Ph.D. degree from Lehigh University. Prior to joining Stockton he held positions at the Center for Naval Analyses, Lehigh University, and the State University of New York at Buffalo. He has held visiting positions at New College of the University of South Florida, Clarkson University, the Software Engineering Institute at Carnegie Mellon University, and the Indiana University (Bloomington) Cooperative Program in Malaysia. He has served as a consultant to major academic, commercial, and government organizations in North America and Asia.

His professional interests include software engineering, artificial intelligence, computer science education, and the mathematics of risk.

THINKING IN Ada - HOW SOME STUDENTS EXPERIENCE THEIR NEW LANGUAGE

Kevin J. Cogan

Program Executive Office
Standard Army Management Information Systems
Fort Belvoir, Virginia 22060

ABSTRACT - It has been said that the way we think is determined by the language which we speak. Translation between spoken languages does not always have a one-to-one correspondence. Computing languages pose the same problem of precisely trying to represent a real-world problem as a computer algorithm. The Ada programming language can be presented to students as having a high correlation with the real-world problem domain. Packages, enumeration, tasking, and exceptions are likely to be new concepts to the student of Ada, but they can be easily acquired through good representative problems. Recognizing the real-world through Ada descriptions can resemble a new natural language for students. Specific course problems and programs which encapsulate the learning experience are described. Through such experiences they will appreciate the power of their new language and enter their careers with confidence.

INTRODUCTION

Linguists have postulated that how we think is determined by the language which we speak. They also note that translation between spoken languages does not always have a one-to-one correlation. A specific example is the German word Gemutlichkeit (a special coziness) without a single English word to precisely represent it.

Computing languages pose the same problem. Software engineers have the difficulty of trying to precisely represent a real-world abstraction in a limited vocabulary. Perhaps the converse situation is more often the case - that the limited vocabulary of a computer language determines the way we think a problem should be represented.

The Ada programming language, with its 62 reserved words, can be presented to students not as a limitation of expression but as a gateway to better and new ways of representing the abstraction as compared to other computer languages. Packages, enumeration, tasking, and exceptions are likely to be new concepts, tools, thought process, and vocabulary. Ada can be an enabling technology for the increasingly challenging software environment.

The existing applications for the space station, air traffic control, and large MIS projects are indicators of the future of Ada and the level of human thought and real-world to computer language translation needed to solve them. Students are told that this is the problem domain for their chosen career. They are told that Ada provides a rich grammar which allows them to exercise their power of thought and expression. Through concrete and successively more complex problems, they acquire a measure of confidence that they will be able to master the problem domain of their future.

This paper draws on over twelve semesters of teaching beginning and intermediate Ada courses to undergraduates. Specific course problems and programs are selected which represent a sample of the student experience in this curriculum. At each introduction of a new Ada tool, the transformation from word to thought is reinforced through a real or futuristic problem. Banquet halls are complex records of various size tables which are themselves arrays with constraints. Soda machines are man-machine interface devices. The post office requires a program to weigh and ship packages automatically. The result of these exercises is an increased ability to conceptualize the real-world in the Ada vernacular. Students are challenged to model a football scoreboard or the instrument panel of their car or stereo. The color code of a resistor offers the opportunity to introduce 'POS and 'VAL attributes in a robotics application to read or paint it.

By mid-semester one can argue that the way the student thinks about a problem has now been altered by the expressiveness of the Ada language. Objects can be stated as a collection of simple and composite types. New tools afforded by the language have elevated the plans on which problems are organized and solved. Student derived term projects are the capstone of the course and serve as a measure of the breadth and complexity of problems that the students themselves feel capable of solving in their new language.

PROBLEMS AND ALGORITHMS

Just like translating from one natural language to another, translation between the real-world and a computer algorithm is not a direct process. A graphic description of this process was presented by Ledgard and Marcotty[1] whereby real-world objects and operations in the problem domain are converted by the programmer into programming language objects and operations in the solution domain. A computer algorithm produces output data which is then interpreted by humans back into real-world objects. From this model one can infer that the higher the level of abstraction permitted by the computer language, the easier one can translate between the real and computer worlds. Students are taught that Ada provides the tools for high level abstractions and that objects can be expressed very

directly. For example, a simple program to mix colors is presented. If the abstraction is to mix blue and yellow to get green than the statement

```
Resulting_Color := Blue + Yellow;
```

is a permissible statement where "+" is an overloaded function for the declared enumerated type Color and the supporting implementation algorithm for the new function "+" is shown in Figure 1.

```
with TEXT_IO; use TEXT_IO;
procedure PALETTE is
  type COLOR is (RED,BLUE,YELLOW,GREEN,PURPLE,ORANGE);
  COLOR_1,COLOR_2,COLOR_3 : COLOR;
  package COLOR_IO is new ENUMERATION_IO(COLOR); use COLOR_IO;
  function "+" (A,B : COLOR) return COLOR is
  begin
        case A is
          when RED => case B is
                        when YELLOW => return ORANGE;
                        when BLUE   => return PURPLE;
                        when others => PUT("MIX NOT DEFINED");
                                       return RED;
                      end case;
          when BLUE => case B is
                        when YELLOW => return GREEN;
                        when RED    => return PURPLE;
                        when others => PUT("MIX NOT DEFINED");
                                       return BLUE;
                      end case;
          when others => PUT("MIX NOT DEFINED");
                         return A;
        end case;
  end "+";
--
begin  --MAIN
  loop
        PUT("ENTER TWO COLORS TO BE MIXED");NEW_LINE;
        GET(COLOR_1); GET(COLOR_2);
        COLOR_3 := COLOR_1;
        COLOR_3 := COLOR_1 + COLOR_2;
        if COLOR_3 /= COLOR_1 then
          PUT("THE RESULT IS "); PUT(COLOR_3);
        end if;
        NEW_LINE(2);
  end loop;
end;
```

Figure 1. Program to mix colors.

The teaching point here is that Blue and Yellow are truly values in an enumerated type and can be as easily computed as one can pour one gallon of paint into another. In another language, say FORTRAN, Blue and Yellow would have to be variable names, converted to a numeric value, computed, looked up in a result table, and Green printed as a character string. This is far removed from the real-world abstraction, and through this kind of example the student quickly learns to appreciate the power of expression permitted by the Ada language.

Students are soon introduced to compound data types through records and arrays as a means to describe real-world objects at a high level of abstraction. An early work by Downes and Goldsack[2] for a hospital patient monitoring system is a formidable case study and discussed with the students. It presents high levels of abstraction and considerable depth of decomposition. Records and arrays composed of other records and arrays permit, for example, the retrieval of the permitted upper blood pressure limit among other factors for the patient in bed 13 of the intensive care unit by using the dot-notated expression

```
Intensive_Care_Unit(13).Safe_Ranges.Upper.Systolic
```

Through this and other such examples, students easily grasp the concept of complex data types where the components of records and arrays can be constructed from other complex types until an atomic level is reached. This permits the student to visualize at a high level of abstraction and develop skills to decompose a problem. Practical exercises described later allow the student to gain experience in this technique. Tools for "industrial strength" programs are early in the making.

LEARNING TO SPEAK Ada

The rudiments of expressing oneself in a computer language are not unlike a spoken language. At the same time that the notions of top-down design, abstraction, decomposition, and parallelism are introduced, the student must also learn from the bottom up the syntax and semantics of their new language as one must learn the spelling, grammar and meanings of a spoken language. It is instructive to observe how a young child with many ideas is frustrated as it struggles to use new words in an unfamiliar grammar. Classroom experience has shown that students need and desire to write real code that compiles and executes at the same time that they are learning concepts, design techniques, and the salient readability, portability, reliability, and maintainability tenets of Ada. A single viewgraph of the 62 reserved words of Ada helps to alleviate any earlier preconceived notion that Ada is a vast and complex language. After all, they are told, only elsif, rem, and xor are not English words and even they are self-evident or require only little explanation. All other reserved words support the high level of abstraction that Ada permits, and through experience, like speaking a language, the actions or semantics they represent will become second nature.

An immersion into the syntax is also immediately called for. A quick walkthrough of Appendix E in the language reference manual, or the equivalent in many textbooks, is necessary. Confidence can be built by a simple introduction to the Backus-Naur Form (BNF) where the meaning of the symbols ::= for denoting a definition, := for assignment, [ ] for 0 or 1 occurrence, { } for 1 or more occurrences, and | for alternation are sufficient to understand and decipher all of Ada's grammar. Students find it not too challenging after all to determine how to verify that the identifier R2D2 is permissible but 3CPO is not, or similarly how 2_000_000 is an optional form of 2000000 for readability, but why .03 is not an allowed decimal number representation. Practice and testing allow students to soon master definitions such as for the case statement.

```
case_statement ::=
  case expression is
    {when choice { | choice) =>
     sequence_of_statements}
```

Such mastery is necessary to test new ideas for abstraction and debug incorrect assumptions for challenging problems later.

AN Ada SAMPLER

Familiarity with the problem domain eases the transition to writing programs in Ada. Consequently, example problems are best understood when they are part of the real-world experience of the student. When introducing the fact that

enumerated types can have values which are overloaded (a value declared in more than one type) this at first may seem foreign or ambiguous. Since most students have taken a general course in chemistry and are familiar with the periodic table of elements, overloading the value Ne for the element neon can be instructive. Neon is both a noble atom

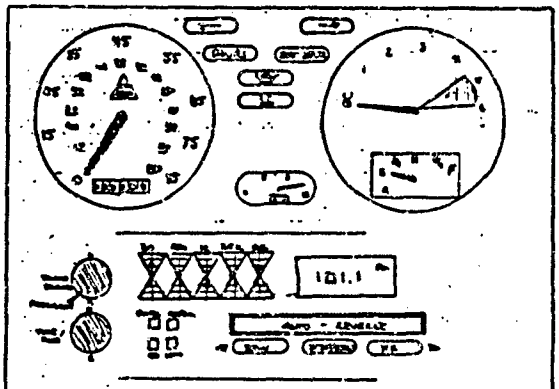
```
type Noble_Atom is (He,Ne,Ar,Kr,Xe,Rn);
```

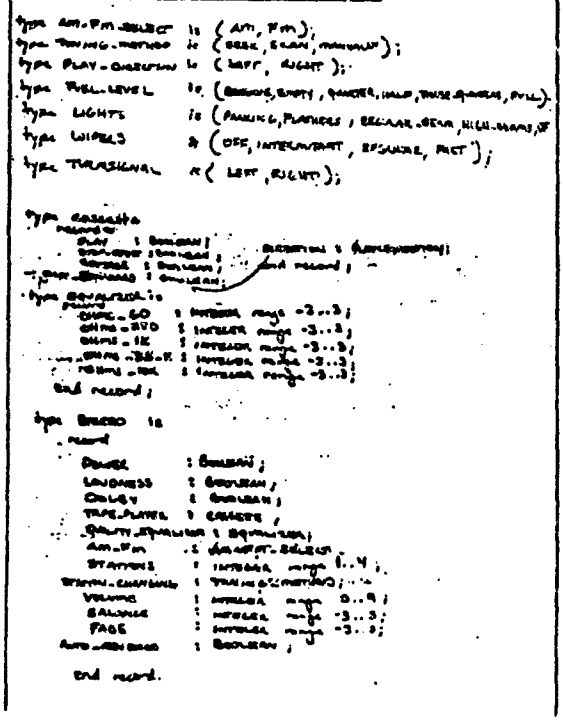
and a period 2 atom

```
type Period_2_Atom is (Li,Be,B,C,N,O,F,Ne);
```

The Deshboard

All students are well acquainted with the automobile. Some are more involved with the technology than others. The instrument panel of automobiles are highly diverse. Some are simple,





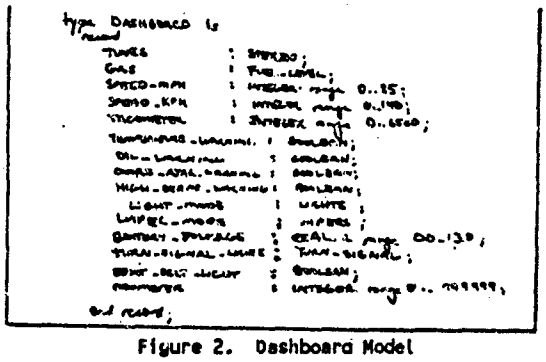
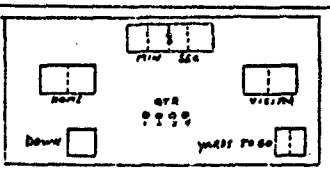


Figure 2. Dashboard Model

some are arrayed like aircraft cockpits. Some are analog, some are digital. Students react very favorably and are often highly challenged to describe as a record in Ada the details of switches, lights, gauges, and knobs on the dashboard of their cars. Taken as a unit, the dashboard is a relatively high level of abstraction, as is the automobile itself. Students, after having been introduced to arrays, records, and numeric types, return interesting and varied homework assignments when given this task. They seem to enjoy (as far as a student can enjoy homework) their ability to express a complex real-world object in Ada and begin to appreciate the translation process between the automotive and computer languages. Figure 2 is an actual student submission for this homework assignment, complete with radio/cassette and graphic equalizer.

Football Scoreboard

With such types of assignments, it is fair to test these concepts on examinations. The football scoreboard design problem tests the ability of the student to model this abstraction also as a record, using arrays with array aggregates and a record for the clock as named components in the record. The problem statement and solution are shown in Figure 3.




```
package SCOREBOARD is
  type TENS_OF_MINUTES is range 0..1;
  type MINUTE is range 0..9;
  type TENS_OF_SECONDS is range 0..5;
  type SECOND is range 0..9;
  type CLOCK is
    record
      TEN_MINUTES : TENS_OF_MINUTES;
      MINUTES     : MINUTE;
      TEN_SECONDS : TENS_OF_SECONDS;
      SECONDS     : SECOND;
    end record;
  type YARDS is array(1..2) of INTEGER range 0..9;
  type SCORE is array(1..2) of INTEGER range 0..9;
  type QUARTER_LIGHTS is array(1..4) of BOOLEAN;
  type DOWNS is range 1..4;
  type FOOTBALL_SCOREBOARD is
    record
      TIME_REMAINING : CLOCK := (1,5,0,0);
      HOME_SCORE,
      VISITOR_SCORE  : SCORE := (0,0);
      QUARTER        : QUARTER_LIGHTS := (TRUE,FALSE,FALSE,FALSE);
      DOWN           : DOWNS := 1;
      YARDS_TO_GO    : YARDS := (1,0);
    end record;
end SCOREBOARD;
```

Figure 3. Football Scoreboard

The Resistor Robot

Sometimes it is useful to supplement the course with unfamiliar subjects and introduce other branches of science and engineering as part of the student experience in the Ada curriculum. Fewer students have taken electronics than chemistry before a programming course. Passing out a handful of resistors and asking what the color code represents can add a new dimension to program design, livens the class, and anticipates the future of software applications. The resistor robot provides such an experience as described in Figure 4.

```
An electronic components manufacturer is converting part of the
operation to robotic applications. Specifically, he wants to
employ a microprocessor-based robot(s) to both read resistor
color codes and convert the code to an integer for sorting, and
input an integer resistor value and convert it to the appropriate
color code for output to the painting machine.

                SIGNIFICANT
  COLOR          FIGURE          MULTIPLIER

 Black             0                    1
 Brown             1                   10
 Red               2                  100
 Orange            3                 1000
 Yellow            4                10000
 Green             5               100000
 Blue              6              1000000
 Violet            7             10000000
 Gray              8            100000000
 White             9           1000000000

with TEXT_IO; use TEXT_IO;
procedure RESISTOR is
   type OPERATION is (READ, PAINT, QUIT);
   type COLOR_CODE is (BLACK, BROWN, RED, ORANGE, YELLOW,
                       GREEN, BLUE, VIOLET, GRAY, WHITE);
   package COLOR_IO is new ENUMERATION_IO(COLOR_CODE);
   package OP_IO is new ENUMERATION_IO(OPERATION);
   package INT_IO is new INTEGER_IO(INTEGER);
   JOB : OPERATION := READ;
   use OP_IO; use INT_IO; use COLOR_IO;
   COLOR_1, COLOR_2, COLOR_3 : COLOR_CODE;
   --
   procedure READ_RESISTOR is
      RESISTOR_VALUE : INTEGER;
   begin
      PUT_LINE("ENTER THREE RESISTOR COLORS");
      GET(COLOR_1); GET(COLOR_2); GET(COLOR_3);
      RESISTOR_VALUE := (COLOR_CODE'POS(COLOR_1)*10
                      + COLOR_CODE'POS(COLOR_2))
                      * 10**(COLOR_CODE'POS(COLOR_3));
      NEW_LINE;
      PUT("The resistor value is ");PUT(RESISTOR_VALUE);PUT(" OHMS");
      NEW_LINE(2);
   end READ_RESISTOR;
--
 procedure PAINT_RESISTOR is
    MULTIPLIER,RESISTOR_VALUE,DIGIT1,DIGIT2:INTEGER;
 begin
    NEW_LINE;
    PUT_LINE("Type in the resistor value ");
    GET(RESISTOR_VALUE);
    MULTIPLIER := 0;
    while RESISTOR_VALUE >= 99 loop
       MULTIPLIER := MULTIPLIER + 1;
       RESISTOR_VALUE := RESISTOR_VALUE / 10;
    end loop;
   DIGIT1 := RESISTOR_VALUE / 10;
   DIGIT2 := RESISTOR_VALUE MOD 10;
   PUT("The colors are ");
   PUT(COLOR_CODE'VAL(DIGIT1));PUT(" ");
   PUT(COLOR_CODE'VAL(DIGIT2));PUT(" ");
   PUT(COLOR_CODE'VAL(MULTIPLIER));PUT(".");
   NEW_LINE(2);
end PAINT_RESISTOR;
--
--
begin --MAIN
   NEW_LINE;
   while JOB /= QUIT loop
        PUT_LINE("Enter a job, READ, PAINT, or QUIT");
        GET(JOB);
        case JOB is
          when READ  => READ_RESISTOR;
          when PAINT => PAINT_RESISTOR;
          when QUIT  => null;
          when others => PUT_LINE("ERROR -- JOB UNKNOWN");
        end case;
   end loop;
end RESISTOR;
```

Figure 4. Resistor Robot Problem

The Automated Post Office

Student program assignments cannot usually be very large given the time available in a one semester course. And yet students should be introduced to topics and experiences germane to Ada. Programming in the large is such a topic. Some day as professional programmers, they could be one of tens or hundreds of programmers on a project. Ada applications can exceed millions of

```
The postal service is giving serious consideration to converting all
of these steps into a 24 hour self-service booth. Conceptually, a
microprocessor driven by Ada software will monitor and operate a
combination of voice recognition systems, scales, optical scanners,
and electromechanical stamping machines and conveyors. The
integration of a complete system will be accomplished as each
process becomes technically available. Therefore, a top-down design
approach will be taken such that new operations can be recoded with
minimal recompilation of existent modules.

An Ada package specification standard has already been decided upon
and it MAY NOT BE ALTERED in any way. It is given as follows:

 package POST_PARCEL is
    type MODE_OF_SHIPMENT is (SURFACE,AIR);
    type REAL is new FLOAT;
    type PARCEL_DATA is
       record
          WEIGHT   : REAL range 0.0..50.0;
          ZIP_CODE : INTEGER range 0..99999;
          SHIPMENT : MODE_OF_SHIPMENT;
       end record;
    procedure GET_COST (DATA : PARCEL_DATA; COST : out REAL);
 end POST_PARCEL;

 package body POST_PARCEL is
 --
 procedure GET_COST (DATA : PARCEL_DATA; COST : out REAL) is
    --
    --
    ORIGINATING_ZIP_CODE    : constant INTEGER := 07305;
    ORIGINATING_FIRST_DIGIT : constant INTEGER := 1;--First digit

    ENDING_FIRST_DIGIT      : INTEGER;
    AIR_CHARGE              : constant REAL := 9.5;
    DIFFERENCE              : INTEGER;
    FACTOR                  : REAL;
 begin
    ENDING_FIRST_DIGIT := (DATA.ZIP_CODE / 10_000);
    DIFFERENCE := ABS(ORIGINATING_FIRST_DIGIT - ENDING_FIRST_DIGIT);
    case DIFFERENCE is
       when 0 => FACTOR := 0.10;
       when 1 => FACTOR := 0.15;
       when 2 => FACTOR := 0.17;
       when 3 => FACTOR := 0.19;
       when 4 => FACTOR := 0.20;
       when 5 => FACTOR := 0.23;
       when 6 => FACTOR := 0.25;
       when 7 => FACTOR := 0.26;
       when 8 => FACTOR := 0.29;
       when 9 => FACTOR := 0.31;
       when others => null;
    end case;
    if DATA.SHIPMENT = SURFACE then
       COST := DATA.WEIGHT * FACTOR;
    elsif DATA.SHIPMENT = AIR then
       COST := DATA.WEIGHT * FACTOR + AIR_CHARGE;
    end if;
 end GET_COST;
 end POST_PARCEL;



 with TEXT_IO; use TEXT_IO;
 with POST_PARCEL; use POST_PARCEL;
 procedure MAIN is
   ZIP_CODE_ERROR, WEIGHT_ERROR, SHIPMENT_ERROR : exception;
    DATA_IN : PARCEL_DATA;
   COST_OUT : REAL;
   DATA : FILE_TYPE;
   package INT_IO is new INTEGER_IO(INTEGER); use INT_IO;
   package REAL_IO is new FLOAT_IO(REAL); use REAL_IO;
   package MODE_IO is new ENUMERATION_IO(MODE_OF_SHIPMENT); use MODE
   function GET_DESTINATION return INTEGER is separate;
   function GET_WEIGHT return REAL is separate;
   function GET_MODE return MODE_OF_SHIPMENT is separate;
 begin
   OPEN(DATA, IN_FILE, 'LAB2.DAT');
   SET_INPUT(DATA);
   loop
     begin
       DATA_IN.ZIP_CODE := GET_DESTINATION;
       DATA_IN.WEIGHT   := GET_WEIGHT;
       DATA_IN.SHIPMENT := GET_MODE;
       GET_COST(DATA_IN,COST_OUT);
       PUT("The cost of parcel is $ "); PUT(COST_OUT);
       NEW_LINE;
     exception
       when ZIP_CODE_ERROR => DATA_IN.WEIGHT := GET_WEIGHT;
                              DATA_IN.SHIPMENT := GET_MODE;
       when WEIGHT_ERROR   => DATA_IN.SHIPMENT := GET_MODE;
       when SHIPMENT_ERROR => null;
       when others         => CLOSE(DATA);
                              exit;
     end;
   end loop;
 end;
```

Figure 5. Automated package mailer.

lines of code and teams of programmers will be needed to write this code. A single project could employ many subcontractors and could be geographically dispersed hundreds or thousands of miles apart. Small programs to teach this point can be used in the classroom. The post office problem is designed to describe the functionality of a new package mailing machine. The package specification is provided by the instructor. It provides the interface between a student writing the application code in procedure called "main" and the student writing the implementation of the package specification in the package body. Separate compilation, compilation order, information hiding, and programming team independence are the learning experiences here as presented in Figure 5.

Rendezvous with the Soda Machine

Parallel processing with Ada's task facilities is a new dimension in programming for almost all undergraduates. It is perhaps the most conceptually challenging learning experience in an Ada course. Six of sixty-two reserved words in Ada are devoted exclusively to the task mechanism. Yet the notion of parallelism is manifested all around the real-world. Bank tellers are "serving" tasks, customers are "user" tasks. Sometimes a "rendezvous" takes place between the two or sometimes such a communication is "guarded" or "timed." There are queues of customers and multiple service requests. The new vocabulary of task, abort, entry, accept, select, and terminate takes on meaning and the BNF tools decipher the grammar for definitions such as

```
selective_wait ::=
  select
    [ when condition => ]
        select_alternative
  { or [ when condition => ]
        select_alternative }
  [ else
      sequence_of_statements ]
  end select;
```

Once again a familiar situation puts the student more at ease when tackling a new language construct. After all, humans can be represented as tasks and so can machines. Students encounter soda machines daily and sometimes hourly. Pushing a button to make a selection is a common man-machine interface. It is a rendezvous with a machine task which "accepts" a selection or guards a selection "when" the beverage choice is empty (a status light is illuminated to highlight this condition). Soda machines may even be smart in the future and communicate the time to restock to the vendor via a modem. Allow the student to conceptualize this notion of a parallel process, add a bit of fiction, and the student may often embellish the minimum homework requirement. The written statement of the problem and a solution is at Figure 6.

```
PROBLEM:

    A cold beverage machine often sits idle even when empty.
All selections generally have a light which is lit when a
particular selection is not available. A customer who chooses
that selection anyway gets a NULL result, and must either make
another selection or no selection. When all lights are lit, the
machine's refrigeration unit and power stays on, and the vendor
restocks the machine on a schedule based on historical data for
rates of consumption. This sometimes results in customer
frustration as well as lost revenues for the vendor if the
historical rates of consumption do not match real demand.

    A "smart" cold beverage machine could reduce frustration,
increase revenues, conserve energy, and govern all of the
machines internal functions. It would employ an embedded
microprocessor for these purposes. It could be coupled to a
communications link which, when sensing that the machine is empty
(all selection lights lit), would notify the vendor's
headquarters of its empty status, block the slot for coin
insertion, and then turn its own power off.
```

```
REQUIREMENTS:

    You are required to emulate the cold beverage machine by
writing a main procedure (parent task) with two sibling tasks,
one which simulates the arrival and customer selection of a
beverage which may or may not be available, and a second task
which simulates the cold beverage machine acceptance of the
customer selection and outputs a message to a file according to
the resulting normal action based of the selection (either
nothing or the type beverage dispensed). Output the selection
light status (boolean) before each customer selection.

    with TEXT_IO; use TEXT_IO;
    procedure MAIN is
      type BEVERAGE is (COLA, ORANGE, GRAPE);
      LAB3_SOL : FILE_TYPE;
      package BEVERAGE_IO is new ENUMERATION_IO(BEVERAGE);
      package EMPTY_LIGHT_IO is new ENUMERATION_IO(BOOLEAN);
      use BEVERAGE_IO, EMPTY_LIGHT_IO;

      task BEVERAGE_CONSUMER;

      task BEVERAGE_PRODUCER is
        entry START;
        entry SELECT_COLA;
        entry SELECT_ORANGE;
        entry SELECT_GRAPE;
      end BEVERAGE_PRODUCER;

      task body BEVERAGE_CONSUMER is separate;

      task body BEVERAGE_PRODUCER is separate;

    begin
      CREATE(FILE => LAB3_SOL,
             MODE => OUT_FILE,
             NAME => "LAB3.SOL");
      SET_OUTPUT(LAB3_SOL);
      BEVERAGE_PRODUCER.START;
    end MAIN;

    separate(MAIN)
    task body BEVERAGE_CONSUMER is
      CHOICE : array(1..11) of BEVERAGE
             := (1|3|5|9    => COLA,
                 2|7|11     => ORANGE,
                 4|6|8|10   => GRAPE);

    begin
      for INDEX in CHOICE'RANGE loop
        case CHOICE(INDEX) is
          when COLA   => BEVERAGE_PRODUCER.SELECT_COLA;
          when ORANGE => BEVERAGE_PRODUCER.SELECT_ORANGE;
          when GRAPE  => BEVERAGE_PRODUCER.SELECT_GRAPE;
          when others => null;
        end case;
      end loop;
    exception
      when TASKING_ERROR => PUT_LINE("BEVERAGE MACHINE NOT AVAILABLE");
      when others => PUT_LINE("Exception raised in CONSUMER task.");
    end BEVERAGE_CONSUMER;

    separate(MAIN)
    task body BEVERAGE_PRODUCER is
      COLA_COUNT,
      ORANGE_COUNT,
      GRAPE_COUNT : NATURAL := 3;
      EMPTY_LIGHT : array(BEVERAGE) of BOOLEAN := (FALSE,FALSE,FALSE);

    begin
      NEW_LINE(3);
      accept START;
      loop
        for INDEX in BEVERAGE loop
          PUT(INDEX); PUT(" EMPTY LIGHT "); PUT(EMPTY_LIGHT(INDEX));
          NEW_LINE;
        end loop;
        NEW_LINE;
        if EMPTY_LIGHT = (TRUE,TRUE,TRUE) then exit; end if;
        select
          when not EMPTY_LIGHT(COLA) =>
          accept SELECT_COLA do
            PUT_LINE("COLA DISPENSED"); NEW_LINE;
            COLA_COUNT := COLA_COUNT - 1;
            if COLA_COUNT = 0 then
              EMPTY_LIGHT(COLA) := TRUE;
            end if;
          end SELECT_COLA;
        or
          when EMPTY_LIGHT(COLA) =>
          accept SELECT_COLA;        -- do nothing
        or
          when not EMPTY_LIGHT(ORANGE) =>
          accept SELECT_ORANGE do
            PUT_LINE("ORANGE DISPENSED"); NEW_LINE;
            ORANGE_COUNT := ORANGE_COUNT - 1;
            if ORANGE_COUNT = 0 then
              EMPTY_LIGHT(ORANGE) := TRUE;
            end if;
          end SELECT_ORANGE;
        or
          when EMPTY_LIGHT(ORANGE) =>
          accept SELECT_ORANGE;      -- do nothing
        or
          when not EMPTY_LIGHT(GRAPE) =>
          accept SELECT_GRAPE do
            PUT_LINE("GRAPE DISPENSED"); NEW_LINE;
            GRAPE_COUNT := GRAPE_COUNT - 1;
            if GRAPE_COUNT = 0 then
              EMPTY_LIGHT(GRAPE) := TRUE;
            end if;
          end SELECT_GRAPE;
          or
            when EMPTY_LIGHT(GRAPE) =>
            accept SELECT_GRAPE;     -- do nothing
        end select;
      end loop;

      PUT_LINE("BEVERAGE MACHINE EMPTY");
      PUT_LINE("VENDOR ELECTRONICALLY NOTIFIED TO RESTOCK");
      PUT_LINE("COIN SLOT BLOCKED, POWER OFF");

    end BEVERAGE_PRODUCER;
```

Figure 6. Soda machine problem.

THE TERM PROJECT

After completing homework problems and laboratory exercises of the type above, students are tasked to choose a term project of their own liking. A series of design reviews are conducted by the instructor who approves the project and the commencement of coding. An interim code walkthrough is conducted, and (hopefully) the student submits a working program by the end of the course which emulates a moderately complex system. Historically, term projects are between 500 and 1000 lines of source code. Some include graphical interfaces. One such project expanded the soda machine with counting change, portraying the status lights changing state, and a soda can dispensed. A partial listing of some projects includes the following system emulations:

Automatic Bank Teller
Crewless Tank
State Lottery Game
Drone Aircraft Target Identifier
Antenna Tuning System
Towers of Hanoi Graphics Game
Fast Food Ordering System
Helicopter Autorotation Simulator

The term projects represent the culmination of what the student has learned, and as importantly, how the student identified with the new language. It manifests how the student now thinks of the real-world in algorithmic form. Time is allocated in the course for students to give an oral presentation and terminal execution of their project to the instructor and the rest of the class. Often there are questions and challenges from the audience. There is a sense of satisfaction with the Ada skills acquired. It is evident during the presentations.

CONCLUSION

The Ada language is an expressive tool for modeling the real-world problem domain. Student can experience their new language in ways that parallel spoken languages through a combination of top down and bottom up design lectures and coding requirements. Ada's close affiliation with high levels of abstraction affords the student with the ability to conceptualize the problem domain and map it to an Ada design. Learning Ada becomes an enabling technology for the student. The language is closely affiliated with the natural way of thinking about a problem. Problem examples, homework, labs, and term projects can provide a fertile test bed for students to experience their new language and build confidence in their programming abilities.

REFERENCES

1. Ledgard, H. and Marcotty, M., The Programming Language Landscape, Science Research Associates, Inc., 1981.

2. Downes, V. and Goldsack, S., Programming Embedded Systems with Ada, Prentice-Hall, Inc., 1982.

BIOGRAPHY

Kevin J. Cogan is a lieutenant colonel in the US Army. He has taught Ada courses since 1982. He was the Ada course director at the US Military Academy and was a part-time associate professor of computer science at Jersey City State College, Jersey City, NJ. He has done research in Ada parallel processing at Rutgers University's Computer Aids for Industrial Productivity (CAIP) center. He earned a B.S. degree from the US Military Academy at West Point and an M.S. degree from Columbia University. He presently serves as the product manager for an Army personnel MIS project (SIDPERS-3) which will represent over 800,000 source lines of Ada code.

Integrating Ada Into Realtime Laboratory Teaching

Dr. Rodney J. Bohlmann
Department of Electrical and Computer Engineering
Valparaiso University, Valparaiso, IN

Abstract

This paper describes a succession of attempts to get Ada teaching started in an engineering curriculum. The target of realtime systems teaching is described. Early experiences with compilers and tools are included as a measure of how to proceed. Classroom experiences are documented with the notable positive and negative results exposed. Curriculum revision to introduce Ada as an enhancement for laboratory teaching is described. Finally the plans to use Ada as a realtime language to target MC68000 based applications are presented.

Introducing Change in Engineering

Change comes hard in Engineering teaching. New ideas are often mistrusted until they have met the test of time. To say it more succinctly Engineering Educators are among the most conservative professionals in the world. They are quick to remark how old-fashioned things are, but are slow to make a really significant change. The changes are left to scientists and a few maverick research and development oriented engineers. Then Engineers are quick to jump on the bandwagon, once the band is in full swing.

In the area of software, the Engineer is suspicious of any price for software until it is shown to solve the tedious problems of design and analysis that are a large part of Engineering. Thus, programs like ANSYS or SPICE can command a high price while a compiler is of little or no value. Further, the development of software is given little or no value by Engineers who are not familiar with the process of software system development. Software piracy on the other hand is not largely practiced among Engineers probably because of their strong adherence to a code of ethics.

Thus for Engineering to embrace Ada as a teaching language was a particularly difficult proposition. Any compiler is viewed as expensive and Ada was doubly or triply so during the early Ada development times at Valparaiso University. Some effort had been spent to move the Engineering Departments into a software design mentality. Until 1979 software was taught pretty much in a syntax and semantics style only, with little or no design process taught. Then the design teaching began and shortly thereafter the switch from FORTRAN to pascal occurred to allow more expressive power relative to the design process.

This author and a close colleague gained a considerable experience teaching the software design process to the entire freshman class of Engineers. From our vantage point in an Electrical Engineering department, the cost of software was seen to take an inordinate bite out of a project budget. This cost was being borne largely because of the inefficient and often ineffective design practices.

As that teaching matured, the Computer Science faculty watched, and after two years of successful teaching with pascal the more mature faculty decided that they could follow suit and change the CS curriculum to the pascal language. Engineering faculty in other departments watched suspiciously until they saw the capabilities of the students emerging from the introductory teaching. The new student was able to design programs and implement them in pascal readily. They soon learned that implementation in FORTRAN was also quite possible from the new design practices.

So some Engineers stayed hooked into FORTRAN even if it meant using the terribly inefficient PC-based FORTRAN compilers. Slowly they maneuvered toward pascal, some actually accepting programs written in pascal in the upper division courses. But, as they shifted, they began to place great worth on the programs written for them by these new students with better capabilities than their predecessors. The faculty started to place a value on the code and so began to "own" it as though it were irreplaceable. In hindsight this should have been addressed as it began. For now the situation became one where a highly conservative group had placed a self determined value on something and while many were attached

to FORTRAN most were hooked on the quality of the pascal programs their students had created for them. Experience is truly the best teacher.

Realtime Systems Teaching Emerges

In the mean time, Electrical Engineering began to develop some good capacity for realtime system work. Realtime in this environment meant that some physical system was connected to a computer and would respond determinably to stimulus from a computer program. Much work was done in C and assembly language and in our laboratory all the work was done using MC6809 processors. Slowly a capacity to do the realtime work in the pascal language was identified as a good idea. A compile time and run time system was developed for programming embedded systems in pascal. This was developed by student and faculty effort and is still a viable system for such work.

During this stage a critical event occurred with the author being appointed to an ASEE summer fellowship at NASA-CALTECH Jet Propulsion Laboratory. During that summer of 1981 Ada was beginning to take off for Department of Defense processes. The supervisor for that summer appointment was Dr. Ed Ng who had been active in the development of Ada and was among its strong proponents within NASA. Needless to say that ASEE appointment had a strong influence to begin looking at Ada as a next step after pascal in the laboratory.

Reading and study soon uncovered some unhappy features of pascal for software engineering. Separate compilation was severely lacking in pascal. It was possible to use pascal include features, but that does not really allow the gains possible from dividing a large program into separate parts for development. While some pascal dialects addressed this deficiency, the realtime development system in use had no prospect for correcting the problem.

The method implemented for realtime control in pascal had been a set of procedures and functions which allowed for addressing of hardware memory locations and fielding of interrupts. The fact that the pascal features were working well was of great benefit to the students. The further fact, that the pascal extension which allowed those features was developed locally, was a serious deterrent to convincing students that high level language was preferred over assembly. After all, the extensions were not easily ported to other pascal compilers. Thus the representation clause, so obviously included in Ada, showed it to be much superior to the pascal in use.

Necessary Early Experiences

The effort to bring Ada into the curriculum then began with an earnest search for a compiler to support our efforts. At the same time the teaching of Ada as a language was planned. A pertinent paper by Jean Sammet pointed out the need to focus on more than the language features [1]. Rather there was a pressing need to move our introductory teaching into a more philosophical direction. This resulted in a formalizing of a taxonomy of sequential program as described in Bohlmann [2].

The compiler search resulted in acquisition of a non-validated compiler running on a PC. The compiler was well documented and the company corrected our difficulties with telephone and updates as needed. R & R Software provided us with a packaged JanusAda compiler in ten copies [3]. This allowed us to provide Ada compilation services to students at our public sites in and around our computer center. The cost of the compiler was borne by the budget in our department, and so needed to be contained preventing us from putting the more robust compilation services on our central time sharing system.

With the effort to teach Ada underway, the limitations of this early compiler were quickly uncovered. The students routinely attempt to use the features of Ada as documented in the Ada Language Reference Manual or LRM [4]. They equally routinely misuse those features leading to some confusion. This is where the study and experience of the faculty is a critical factor. Such problems as small symbol table space, and program and data size limitations on PC's were part of our learning process.

These problems were not unexpected. What is of interest is that the scope of problems that the faculty felt comfortable assigning to students increased dramatically with the prospect of success because of the features of Ada. It was now possible to do a divided development with separate compilation. During the first project the students were divided into groups of two or three. Each

group was assigned responsibility for a part of the design. Class wide planning sessions were used to divide the problem and assign the parts. The next class meeting was devoted to discussing preliminary specifications as developed within the groups. These specifications were evaluated for completeness and service quality to the project. Once the specifications were agreed upon, each team had the assignment to code the specification and compile it into a common library. This step proved to be a pertinent catalyst for the teams to produce body text to support their own specifications.

Though the compiler in use at the time had some limitations, the students were able to progress rapidly into proper design for testability and reusability. These features continue to be important features of software engineering. Further, the difficulty with the compiler fostered a language lawyer attitude among the better students. This attitude was the key factor in extraction of the pertinent semantic features to be applied. As the teams interacted, questions arose about how to express algorithm or data requirements. The language lawyers among us would quickly come up with solutions. In retrospect, that capability was necessary for the success in our efforts. A perfect compiler may in fact not have enabled us to develop that capacity.

This first experience added valuable knowledge into the environment at Valparaiso University. The students felt like they were learning and doing some serious large program work. The faculty felt like they were learning and leading students into the philosophical approach already recognized as the appropriate method of software teaching. While the over all functionality of the first project was never realized, the students who participated uniformly rated that as their best programming experience to date. In the years that have passed, most of those involved, students and faculty, have come to a much larger experience. This was a good start.

Beyond that first project, the JanusAda compiler was used until further budget was allocated to upgrade. One upgrade was purchased and donated to the school by the faculty involved, but finally it was time to search for a new compiler. In the years that had passed the compiler technology had progressed. Several validated compilers were available for PC work. Meridian, now a Verdix company, offered a price competitive deal on a compiler. Looking primarily for a validated compiler, cost was the other factor in choosing the next compiler. Meridian University Support provided us with evaluation copies of their PC-based Ada compilers. The compilers solved all of the problems that had earlier limited our use of Ada. The version 4.0 AdaVantage compiler solved the symbol table problems and the extended memory compiler solved the large program space limits when needed. Not unexpectedly, the new compiler brought some idiosyncrasies of its own.

Shortly after selecting the Meridian Ada 4.0 compiler as the replacement, Meridian announced two products of interest to us. One was their AdaStudent which provided a validated Ada compiler with the limitation that no library linking was allowed. This simply required that all code be compiled into the same library. For small programs and syntactic learning this was superb. Superb because of its price. Our one time fee of $1000 allowed us to distribute ten copies of AdaStudent to our students for use on their personal computers. We also had the larger compilers for the more central sites and so the combination worked well. The second product that appeared then was the AdaZ product which later changed name to OpenAda. The cost of this complete compiler was dropped to $149 and several students and faculty opted to purchase their own compiler. This compiler had an integrated environment for editing and compiling with language directed edit functions. It showed some direct competition for functionality with the Turbo Pascal product in use in our freshmen course.

Continued use of the compiler started to uncover some of it failings. During one process of development we discovered that dynamic memory unchecked deallocation was dysfunctional. We also experienced some problems with order of the compilation of stubbed sub-programs. The most recent upgrade solved the problem of memory control and as of this writing compilation order problem is being investigated by Meridian. Even with the problems encountered the Meridian compilers served us and continue to serve our needs in a more than adequate way.

Classroom Experiences to Build on

The first of the classroom projects attempted with Ada centered in a course on

simulation. Past work on event driven simulation provided a platform for developing an event driven simulator for digital systems work. The goal of the project was to apply Ada to a real problem that had testable results.

This project afforded the experience of separate compilation, with team efforts for specification and body development, and with language lawyer attitudes as described earlier. The basic concept of a generic insertion queue was a principle feature of this work. Students were delighted to learn of generic capacity to allow for full featured programs.

Implemented in R & R JanusAda the project faltered with the large data needs and expansive programs created by novice Ada programmers. A good experience allowed for further work on Jades with a later group of students. It is a manager's responsibility to maintain the documentation and code for such a hiatus in activity. The one year layover was managed by the faculty and allowed for even more programming learning as the problem was resurrected later.

Creative Course Development

Teaching in this way was not a routine thing. The courses on Ada were housed in a catchall *Topics in Electrical and Computer Engineering* title. The Ada topic was germane to the course with different problems chosen by students. Another feature of this course is that it is not a fixed number of credits. In the four semesters that Ada was taught under this rubric, semester credits of one, two, and three were all used for student records. Each student agreed on a level of work with credit commensurate with the level. Projects allowed for varied participation and students were able to fine tune the workload and credits to their curriculum needs. Minimum requirements of simple programs ensured basic knowledge for the poorest performing students. This topics course is documented in the CREASE 6.0 catalog [5].

Experience behind us, a proposal was prepared for the College of Engineering to do an experiment with our freshmen course, *Introduction to Algorithms for Computing*. This course had a rich history of teaching with clear goals and syllabus. It had begun with FORTRAN and progressed to pascal. The textbooks used began with Tremblay and Bunt [6] for the first few years and then shifted to Finger and Finger [7] for a little better Engineering flavor. Students worked with central time sharing computers and at the best times, with selected students, projects were as complex as image processing and separate modular compilation. Turbo pascal came into vogue toward the end of life of the course and became the compiler of choice because it supported the Ada-like feature of units.

The proposal was to take that course, select a group of students, and in one section of five, convert the course to the language Ada. In concession to the FORTRAN users the pascal sections used a textbook with some FORTRAN in it. The Ada course used a well written text by Putnam Texel [8]. This book introduced package as the base concept and built on it throughout.

Projects in the Ada course were chosen to exemplify Ada and at the same time parallel the projects in the control sections. Exception handling for input output processes, instantiation of generic input output routines, overloading and other features of Ada were necessary introductory material covered in projects. A final project using a graphics utility library was attempted. This however faltered because of failures in the graphics package.

Students were selected based on a self selecting process. Experience in high school with programming other that Basic was required. Predisposition to a computer intensive Engineering curriculum and career was suggested. With this simple guideline and some advisor assistance a group of 17 freshman were identified as the Ada guinea pigs.

The performance for programming and the perceived capability of the students after the course was positive. The philosophical approach that had been tried and true in the previous pascal courses, excelled in the new Ada section. The performance of the students in the following course would be the test of how well the approach actually worked.

Evaluating the Experiment

The second course on Algorithms, taught outside of Engineering, in the Mathematics and Computer Science Department followed in the immediate next semester after the Introduction

course. This course enrolled students from three introductory courses. First, students from the introductory CS course were enrolled. Secondly the pascal sections of the introductory engineering course fed students into the second level algorithms course. And now for the first time the third group, students whose first experience was in Ada, were enrolled with the others.

The pascal language was strongly entrenched in this Algorithms II course and posed some challenge for the Ada students. While the Engineering faculty felt they were prepared to continue their studies, the CS faculty balked, unwilling to do comparative studies with two languages. The Ada students requested help sessions for the parallel comparisons allowing them to do pascal work. The author agreed to the help sessions under the proviso that the CS faculty attend to guide them in the right direction. At first this felt like the Ada teaching had failed to be broad enough. However, as the help sessions progressed the students themselves took over the study, and in the manner that I had hoped for, they used the philosophy that they had learned studying Ada to extrapolate the pascal needed to excel. The CS faculty also began to get a positive exposure to Ada as the students would conclude that they needed to find a "different" method to do in pascal what they knew readily how to do in Ada. As they worked out the detail they would implement their ideas in Ada and then study how to implement the same algorithms in pascal. The CS faculty finally accepted the Ada solutions as solved problems and the Ada students were successful in the course.

Success Builds Interest

A continued effort for topics courses allowed for a wide variety of experience with Ada. Among the first topics to be studied in detail was tasking. This was chosen because of the strong interest in realtime multi-processor applications. Applications such as this are a regular part of the production systems in the steel mills in Northwest Indiana. As a consultant, the problems encountered were largely with the tasks and interprocessor communications when multiple processors were involved. Thus, some study in the area of tasks was of interest.

One student in the tasking topic, chose to implement an array of tasks to solve the classical Towers of Hanoi problem. He chose to use block graphics for a two dimensional representation of the solution as it progressed. The general organization of his solution was to make each ring a task and each channel for moving a ring on the screen a task. The rings were then free to move as the program could identify a place for a ring to move and allocate a channel task to carry the ring. The Ada source code is not particularly difficult and is available from the author.

Another topic dealt with extending the functionality of Ada to matrix operations. Students were assigned the problem of creating a set of overloading **get** and **put** routines to **get** and **put** vector and matrix data. An interesting note here is that the students were all taking Ada as a second language and were weak in there notions of overloading. Though we had discussed overloading in class the first cut solutions of all but a few students failed to use the procedure names of **get** and **put** to overload the generic operators with ones having vector or matrix parameters. After a brief discussion in class, the students were quite receptive to the suggestion that the proper solution was to use the names **get** and **put** as overloaded operators. A further feature suggested by students that completed the input output package, added the parameters standard to the generic input output procedures for numeric types. The fore, aft, and exp parameters were added as was a parameter to control the display of indexes as output was presented. Default values were included to make the routines conform as much as possible to the generic input output routines documented in the LRM.

The prompting index operation was most useful for interactive output of array data. Since much of the programming done was for interactive use, the prompt was selected by students as an important feature. Thus, the overall package for vector and matrix input and output was implemented and the students were then assigned the problem of creating a matrix and vector calculator. Students less able to create such a program were allowed to do a sequence of **get** and **put** operations with computations in between. The matrix and vector operations were acquired in a matrix package authored by Dr. Roger Lee and moved by ftp from the Simtel20 Ada repository [9].

During the time that the matrix input output operations were under development, another section of students in the same semester accepted

the assignment of perusing the network archives for Ada packages of use to our programming and teaching. Though Valparaiso University was using a limited BITNET connection at the time the resourceful students were able to discover and retrieve several useful artifacts. For graphics interactions a set of mouse drivers in Ada was retrieved and these same students also retrieved the array operations package for their fellow students in the other section. One of the special features of Ada is that the packages and others that followed were directly compiled and linked into our developing system. This was a rather important point for students who were just starting to see the big picture in large program development. They realized that the programmer does not necessarily need to create all of the code for a project. Some code can be acquired by network or purchased from vendors to the job done more rapidly. With this also comes an awareness for software license and propriety of use.

Brief comments are in order for two other Ada projects done in the topics area. The first was done for a student who also was a bicycle designer and fabricator. The students in the section with him agreed to create a bicycle design program to select from standard parts and create a bicycle. Special parts were also included to allow for custom frame sizing and custom wheel development and such. The overall success of that project was limited. Students working in that section were not as well prepared to take on an Ada task at the beginning of the semester and almost half of the semester elapsed before the program design could begin. Starting this late with a design for the program, the code never reached beyond multiple group specifications being compiled successfully. Implementation of only a few of the bodies limited the testing of the program to only a very few functions. Students, even so, reported confidence that the large program they had started was well designed and on the brink of completion.

The other project undertaken was to attempt to create an Ada package for a DASH-8 and DASH-16 data acquisition system embedded in a PC. The approach by this sophomore group was to first choose between creating new code and translating the original pascal drivers. They opted to translate the pascal drivers, largely because of their familiarity with pascal and less familiarity with hardware features of things like data acquisition systems. Again the completion level was limited but the student satisfaction was high.

A New Approach with a New Tool

With the third year of topic teaching a new software product was acquired. This tutorial product, NSite Ada [10], allowed for students to study topics in Ada in a computer directed manner. The limited budget however, put some constraints on the use of the built-in examining functions. Students needed to record their own quiz scores and keep track of their own progress. Even so the students progressed rapidly into a good knowledge of Ada and software engineering. The directed learning was an excellent approach to instilling the philosophy that had come to be a part of our teaching. The material itself was organized into chapters that followed the organization of the LRM. The students studied the material in sequence from start to finish.

Often times the students would get together as groups and pursue a chapter in the tutorial material quizzing and discussing the matters as they went. They then would jointly solve programming assignments and experiment with new ideas based on the programs completed. The style of learning which resulted was superior to the lecture and discussion style that had been in use. With this style the interaction between faculty and student moved to a joint investigative phase. Now students could assist in raising the level of language understanding which the faculty was pursuing. The more rapid acquisition of language capability by the students also allowed for more direct programming assignments.

The NSite product, as first acquired serves quite well despite a significant number of typographical and content errors in the tutorial material. The NSite tool is at this time under re-evaluation at Valparaiso as we consider a competitive product LearnAda [11] which covers the same need. Independent of which tutorial product is used, the computer based tutorial will provide a good method of introducing the language. It also provides a rapid access to reference material both in the tutorial text and by having a copy of the LRM on line.

With students studying Ada by tutorial, classroom time was free to pursue specific programming topics. One topic that was studied as a means of understanding the Ada run time support

package was task preemption. The Meridian compiler in use provides for task preemption and so run time controlled time slicing and priority assignments for tasks were usable. Simple programs were written that displayed time slicing and task priority behavior with preemption enabled and with preemption disabled. The results were not surprising for the experienced faculty, but were powerful examples for students just starting to consider how a computer could be used in a multi-tasking manner. The amount of time required to do the example programs with the class would not have been available if the class time would have been used for Ada information lectures. The computer based tutorial enabled these studies.

A Project That Works

Experience builds confidence and so it is in the case of experiences using Ada. The philosophy of software engineering and the use of Ada became ingrained with the faculty. As Ada-9X promised to come into existence, the experience with Ada-83 was continuing. During the Fall 1992 semester the most recent assignment was made to a group of Computer Science students studying Ada under Engineering faculty supervision. These students were given the assignment of building a part of the old JADES project in a new way. In order to do event driven simulation an ordered queue of events to be processed needed to be supported. The assignment given to the group was one of building an Insertion Queue which contained elements and invoked a user supplied order relation to insert new items in the Queue for later removal in order. This has direct application for sequencing events during simulation.

The creation of a generic procedure to provide insertion queue capability proved to be the most interesting assignment to date. The technical task was not that difficult. What was interesting is that the students were Junior and Senior level and had a deeply ingrained pascal flavor to the their style. Specific discussion about overloading, generic objects and instantiation details was a regular part of class meetings. Students began the semester with pascal knowledge and used the NSite tutorial to gain Ada knowledge. It took about five weeks before the final form of the generic queue could be envisioned by the students.

Once they had a clear view of the problem it was divided into three parts with Enqueue, Dequeue, and QueueSize naturally dividing the problem. The procedure Enqueue and Dequeue and the function QueueSize were created as stubbed separate bodies. This allowed for complete syntax free compilation prior to finishing the detailed algorithm. It also allowed for a complete replacement of the internal storage method for the sorted queue, should performance become an issue in the use of the Insertion Queue. Students were divided into three groups to complete the stubs and the whole package was brought together with a test program four weeks before the end of the semester.

Testing uncovered a problem with unchecked deallocation which was corrected by a compiler upgrade which was purchased earlier and had arrived about a week before it was needed. Some call that just-in-time delivery while in our case it was just-dumb-luck timing. Another matter that was uncovered in testing was discovered as the separate bodies were compiled. The library manager does not update the body executable code for a separate body when that body is compiled after the parent unit. The order of compilation is being looked into by Meridian and we expect that it will be resolved cleanly. In the mean time we can assume that what we are trying to do may be outside the scope of the compiler and constrain our compilation order to not recompile separate bodies without first compiling the parent.

The Insertion Queue was completed and tested with a variety of queue element types and differing order relations. It includes the exceptions Full_Queue and Empty_Queue. This package is the first generic package at VU that was completed well and with full functionality and good documentation. The assignment was smaller than previous assignments, but the problem was solved in a more elegant manner than previous solutions. Finally, the tools of Ada were available to the students because the faculty had enough grasp of the language to coach them properly.

Curriculum Revision to Include Ada

As the Ada language experience developed, the curriculum at Valparaiso University was under revision. Four disciplines of Engineering worked to create a curriculum that would best serve each. As part of that work a serious review of programming teaching was done. Within Computer Engineering, Ada was selected as the language for realtime embedded systems teaching. This lead to a decision

to supplant pascal from the early curriculum and use Ada as the first language. It also results in an impact on the second and third year programming courses that need an Ada revision.

The other disciplines of Civil, Mechanical and to a lesser extent Electrical Engineering reached the conclusion that teaching about programming was much less important in their new curriculum. This reflects the thinking that programs needed for discipline specific activities are available commercially, and little or no programming will be needed to apply them correctly. There is even the sentiment that when programming is required it can be employed from knowledgeable persons on an as needed basis.

Those conclusions might be better understood if one reviews the results of the curriculum discussions. Pascal was deeply entrenched and pascal tends to make programming easy as compared to using Basic or FORTRAN. Thus the Engineering professors were in a better situation with pascal than they had ever been. Secondly, there was a strong sentiment that FORTRAN should be taught to engineers. The arguments for FORTRAN reduced to "There are a lot of FORTRAN programs running." and "Most of the other schools are doing it." While these arguments are irrefutable, they do not, in the authors opinion, warrant continuing with a language that has successfully completed its service.

There also surfaced a sentiment that Valparaiso University College of Engineering should not be the leader in the area of Engineering Software teaching revision. Noting that, the decision was finalized that Computer Engineering faculty will teach programming for Computer Engineers and Electrical Engineers and that Civil and Mechanical Engineers will take computer literacy courses specifically about software in their discipline. This result is now being implemented and for the time being seems to be the correct decision for our school.

Thus the Computer Engineering course, *Algorithms for Computing*, based on Ada, will be taught for the first time in the fall of 1993 to the first wave of Sophomore students studying in the revised curriculum. They reach this first course as Sophomores because we chose to introduce hardware concepts first in the second semester Freshman year immediately after the major is declared in Computer Engineering. Using logic design as the basis, it is expected that benefits will accrue for software engineering teaching in the first programming course.

Directly coupled to all of the Ada development is the decision to use Ada to teach embedded systems design. The Computer Engineering laboratory sequence and two elective courses address the issue of high level language usage for embedded systems in various ways. Pascal was in use for ten years and served well as described earlier. With the MC6809 processor, and a locally extended pascal native code compiler, students were able to create significant products for course work and for sponsored development projects. In one case, the students were able to demonstrate the value of high level language development over assembly language for a small company. The dramatic decrease in expected development time was a convincing experience for the project Engineer.

With pascal working so well, it was no easy decision and is no easy task to work toward a major technology shift. Yet, with the better features found in Ada and the better cpu capacity of the MC68000, the time was right to make some changes. The vendor support from Motorola and from Ada compiler vendors is growing as we make the necessary effort of contacting them and exposing our needs. This continues to suggest that we are making progress.

The shift to the MC68000 is now complete with the installation of MC68000 based systems for software and hardware development. These systems were built in conjunction with the University of Toronto where they were developed. The features of the boards include 2Mbytes of memory, both AUI and BNC ethernet ports, four RS-232 ports, printer ports and parallel I/O connectors. The network connection allows the boards to be supported from our growing workstation environment.

Presently the programming for laboratory and course work is being done in C and assembly language. Programs are compiled or assembled on a Sun 3/60, an MC68020 based workstation. The compilers and libraries are part of the package received from University of Toronto. Object modules are loaded into the target system by ftp connections via the ethernet. It is also possible to

load object modules from the TTY ports.

While C is not the language of choice, it is the only compiler at VU at this time capable of generating MC68000 code. Negotiations and financing are in progress to place an Ada compiler on the workstations and to develop the methods of compiling Ada code and loading it by ftp connection as is currently being done with C. Such operation is available for purchase and is actively being pursued.

Ada Blossoms

The blossoming of Ada at Valparaiso University has been a long process. Ada is not yet in full bloom though critical junctures are past. Recent agreements assure better funding in the future and support for faculty development to a higher level has recently been funded. The demand for Ada learning continues to grow among our student body. That demand may also grow in the work place as the students and vendors get the Ada message to the managers.

There is no substitute for interaction with colleagues in other places who are doing Ada teaching. Thus, the support for attending ASEET and ANCOST sponsored meetings in past years and the recent support for two of us to attend Tri-Ada is both useful and necessary. The effective return on the investment in our case was greatly enhanced by participation in tutorials at the meetings. Over the years, tutorials by Engle and Dominice [12], Cook and Vega [13], and Rogers [14] were all attended and have contributed significantly to the positive Ada experience.

Now the plans are frozen which will allow us to work toward a goal. The obstacles of development have been removed and the new Ada based, *Algorithms for Computing* course is being readied. The impact on the second level courses is anticipated. The compiler for laboratory use will grow out of the current funding mechanism.

So once again it is time to begin. Begin, because up until now Ada teaching was an experiment which was providing great experience. Now the students' learning and career development will depend on the results. This is a serious matter and we do not take it lightly as we begin teaching introductory programming with the language Ada. Ada-83 is now ten years old. We have followed it and used Ada-83 and as we begin teaching it in our mainstream curriculum, we expect to migrate to Ada-9x easily and painlessly.

Conclusion

Ada knowledge is not easy to acquire. Experience is needed to build the confidence in faculty that Ada is neither bigger nor smaller than other programming languages when beginning teaching is considered. There is a limited amount of information that can be imparted to beginning students. It is important that the information be well selected and well presented to foster a quest for knowledge. With Ada, software engineering and philosophical approaches to programming, Computer Engineering students at Valparaiso University will be quite ready for the future development and application of computer hardware and software.

Bibliography

[1] Sammet, Jean, "Why Ada is not Just Another Programming Language," Communicatons of the ACM, Vol 9, No 8, August 1986, pp. 722-731.

[2] Bohlmann, Rodney J., "A Taxonomy Approach to Introductory Programming", IEEE Transactions on Education, February 1993.

[3] R. R. Software, JANUS/Ada Compiler for 80x86 Computers, Madison, WI.

[4] ANSI/MIL-STD-1815A, *Ada Programming Language*, January 1983.

[5] ***CREASE 6.0*** *Catalog of Resources for Education in Ada and Software Engineering*, Ada Information Clearinghouse, Lanham, MD, February 1992.

[6] Tremblay, Jean-Paul and Bunt, Richard B., *An Introduction to Computer Science an Algorithmic Approach*, McGraw Hill, Inc., 1979.

[7] Finger, Susan and Finger, Ellen, *Pascal with Applications in Science and Engineering*, D.C. Heath and Company, 1986.

[8] Texel, Putnam, *Introductory Ada*, Wadsworth, Inc., Belmont, CA, 1986.

Bibliography, Continued

[9] Lee, Roger, Matrix_Package, *Version 1.0*, Naval Air Development Center, Advanced Software Technology Division, Warminster, PA, February 1984.

[10] Network Solutions, Integrated Training Environment for Ada, NSITE-Ada, *Version 2.1*, Herndon, VA, December 1990.

[11] Aetech, *LearnAda for MS-DOS*, Carlsbad, CA, 1992.

[12] Engle, Charles B. Jr. and Dominice, Tony, *Ada from a Management Perspective*, and McKee, Gary, *DoD-Std-1838 and other Environment Interfaces*, Fourth Annual ASEET Symposium, Tutorial III, Houston, TX, June 1989.

[13]Cook, David A. and Vega, David, *Ada Tasking and Generic Program Units*, Sixth Annual ASEET Symposium Tutorial, Alexandria, VA, September 1991.

[14] Rogers, Patrick. "Realtime/Embedded Systems and Ada", Volume 1, Tutorial 3, Tri-Ada '92, Orlando, FL, November 1992, pp. 105-260.

About the Author

Dr. Bohlmann is an Associate Professor of Electrical and Computer Engineering at Valparaiso University. He specializes in realtime systems and teaches courses in the areas of computer architecture, computer networks, and VLSI design. He is a Professional Engineer in Indiana, a member of IEEE and IEEE Computer Society and is the Vice Chairman of the Calumet Section of IEEE. He is also a member of ACM and ACM-SIGAda. A consultant in realtime systems work and microprossessor development, Dr. Bohlmann may be reached by e-mail as rjb@gellersen.valpo.edu.

NEXT GENERATION COMPUTER RESOURCES
(NGCR)
PROJECT SUPPORT ENVIRONMENT STANDARDS
(PSES)

Tricia Oberndorf
Naval Air Warfare Center
Aircraft Division Warminster
Code 7031
P.O. Box 5152
Warminster, PA 18974-0591

Carl Schmiedekamp
Naval Air Warfare Center
Aircraft Division Warminster
Code 7033
P.O. Box 5152
Warminster, PA 18974-0591

LCDR Vincent Squitieri
Space & Naval Warfare Systems Command
SPAWAR 2312B4
Washington, D.C. 20363-5100

SUMMARY

The U.S. Navy has embarked on the Next Generation Computer Resources (NGCR) program to fulfill its need for standard computing resources. The program revolves around the selection of interface standards in six areas. One of these areas is project support environments (PSEs). The projects supported by the environments develop, enhance or maintain computer-based systems or products. These interface standards should be useful for projects focused primarily on software development, hardware development or the concurrent development of hardware and software. They will include support for Ada. This paper discusses the approach and plans for the selection of these PSE standards and the issues that must be addressed for the effort to be successful.

INTRODUCTION

The U.S. Navy has embarked on the Next Generation Computer Resources (NGCR) program to fulfill its need for standard computing resources. The program takes an open systems architecture approach and revolves around the selection of interface standards in six areas. One of these areas is project support environments (PSEs). The projects supported by the FSEs develop, enhance, or maintain computer-based systems or products, including those in which Ada is used. These interface standards should be useful for projects focused primarily on software development, on hardware development or on the concurrent development of hardware and software.

BACKGROUND

NGCR

The Navy has a long history of developing and using standard computer products. When computer technology was in its infancy, the Navy wielded significant influence in the market, setting its own requirements and developing its own computer designs, including Instruction Set Architectures (ISAs). Standard computer implementations (i.e., buying "boxes") and upward compatible ISAs have been the foundation of the Navy's computer policy.

This policy has been motivated by the fact that software can adapt a common computer design to meet many different applications.

But the Navy's current computer standardization approach is not technologically competitive in today's environment of rapidly improving technologies. The Navy acquisition and budget process takes a long time to field new standard computers, so long that the produced technology is often old (compared to commercial technology) by the time it is fielded. The obvious logistics benefits associated with standard hardware are offset by the inability to field current technologies. In addition, the DoD in general no longer is a major factor in the marketplace and cannot dictate what it wants or needs. The fact that the defense budget is shrinking means that the Navy can no longer afford to go it alone and instead needs to leverage off of the commercial marketplace.

The objective of the NGCR program is to restructure the Navy's approach to acquisition of standard computing resources to take better advantage of commercial advances and investments. It is expected that this new approach will result in:

. reduced production costs (through larger quantity buys)

. reduced operation and maintenance costs

. avoidance of replication of Navy RDT&E costs (for separate projects to develop similar computing capabilities) and

. more effective system integration.

The proposed new approach is an open systems approach based on the establishment of commercially-based interface standards in six areas: multisystem interconnects, multiprocessor interconnects, operating systems, database management systems, graphics standards, and project support environments. This open systems approach is consistent with the trend throughout the industry.

The NGCR interface standards will be based on existing industry standards with multi-vendor support. In cases where existing industry standards do not fully meet Navy needs, the approach is to further enhance the existing standards jointly with industry. This will assure the Navy of a widely-accepted set of commercially-based interface standards.

Application of these interface standards will change the Navy's approach from one of buying standard computers to one of procuring commercial computing resources which satisfy the interfaces defined by the standards. These standards will be applied at the project level rather than a Navy-wide procurement level.

PSESWG

The effort to establish the PSE interface standards was initiated at the start of 1991 with the inaugural meeting of the Project Support Environment Standards Working Group (PSESWG - pronounced "peace-wig"). As with all of the NGCR working groups, this is a joint industry/academia/government group of technologists with backgrounds in the requirements for and issues regarding PSEs. This Navy-led group is committed to the identification of PSE interface standards in the form of a military standard, with an accompanying military handbook, by 1998.

APPROACH

Some of the approach taken by the PSESWG is dictated by the NGCR program. Other aspects are driven by the necessities of the PSE area. The following are the key elements of the PSESWG approach.

1. Joint industry/academia/government working group

All of the NGCR standardization efforts are accomplished by working groups with strong industrial, academic and government participation. The PSESWG is well-balanced, including members from the research community as well as the PSE user community. It draws heavily on industry expertise as well as that available from all facets of the government. All three services, the federally-funded research and development centers, and the National Institute of Standards and Technology (NIST) have been represented at meetings. This balance provides technological strength while assuring the group that there is a sound balance between the perceived Navy requirements and the directions in which industry is heading.

2. Standardization on interfaces, not products

One of the keys to the NGCR program is to get away from standardization on products and to move towards open systems. To achieve this means that the emphasis must be on interfaces, services, and protocols, not specific vendor implementations, even if those implementations have been created according to Navy specifications. There are plans to maintain Certified Product Lists for some of the NGCR standards, and it may be appropriate at times for the Navy to make large procurements of products that meet the standards. However, the important thing will always be the interfaces rather than the products. In this way the Navy can enjoy the benefits of new technology and increased interoperability of its systems.

This general NGCR principle applies to the PSE effort as well. It is sometimes difficult for PSE users to think of their systems in terms of the interfaces between the components. But doing so is key to being able to include the best of the available tools in Navy project PSEs and to achieve productive integration of the PSE components, even when they are from a variety of vendors.

3. Selection of existing industry standards

Another emphasis in the NGCR program is on the ability of the Navy to become a part of the industry marketplace, thus being able to take advantage of industry innovations and advances in technology. To achieve this, the program focuses on the selection of existing industry standards whenever possible. We will often find that existing industry standards do not quite provide all of the features required by the Navy, so it is common for the working groups to become active in the industry organization responsible for a standard that has been selected. If there is a firm requirement for an interface for which no viable industry standard can be found, the working group will decide whether the need is strong enough to try to find industry interest in converging on such a standard. Only as a last resort will a working group create its own unique interface; it will often be preferable to simply defer the requirement until an industry standard emerges.

For the PSESWG this is even more straightforward than for the other groups. There is most often little to distinguish a DoD PSE from that required in any other setting. Small parts (e.g., the requirements to support Ada) may be different, but they do not affect the essential nature of a PSE nor most of the interfaces needed to support PSEs. The final PSE standard is expected to refer to new and existing environment interface standards and to be usable in the procurement of Navy (and other) systems in 1998. The initial focus of the PSESWG is on identifying those areas of support environments that should have standardized interfaces and for which industry-accepted interface standards can be available within the project's time frame.

4. PSESWG organization

The PSESWG has been organized into subgroups and teams. The subgroups are: Reference Models, Available Technology, and Approach. The Reference Models Subgroup is working in cooperation with the NIST Integrated Software Engineering Environments (ISEE) effort to produce a full environment reference model. PSESWG intends to use the model for identifying PSE interface requirements and describing PSE technology. The Available Technology Subgroup is collecting and reviewing descriptions of existing environment interface technologies. The Approach Subgroup is planning the organization and operation of PSESWG, including procedures for the selection of baseline standards.

The PSESWG has recently organized teams which are more focused upon standard production and specific technology areas. (PSESWG members generally participate on one subgroup and one team.) The initial teams are Data Interfaces, Framework, and Standard and Handbook Writing. The Data Interfaces Team is tasked with investigating the data interchange technology area and its subareas, producing an interface requirements document for the technology, and producing a list of candidates for selection as part of the developing standard. The Framework Team has tasks similar to those of the Data Interfaces Team for the framework technology area. The Standard and Handbook Writing Team is tasked with actually writing the draft military standard and draft military handbook.

5. PSESWG Standard

The Project Support Environment Interface (PSEI) standard will not define standard tools or tool sets for use in Navy system development. Instead, the focus is on tool integration mechanisms (including frameworks), data exchange mechanisms, and the logical contents of project data repositories. An integrated (harmonized) set of environment interface standards is important in the success of NGCR.

Technically, the adoption of standards for PSE interfaces, services, and protocols will provide a means for better integration within a PSE and better interaction between different PSE implementations. Procurement of PSEs will be aided by making their specification easier and by lowering costs for common PSE components.

PROGRESS AND ACCOMPLISHMENTS

PSESWG effort to-date has focussed on three main products[1]: the Available Technology Report, the PSE Reference Model, and the initial selection of some PSE-related standards.

Available Technology Report

This report is a compendium of technologies that are available today that are believed to have relevance to the PSESWG effort. Most of the entries describe interface standards, both recognized and *de facto*, although a few describe products that appear to address aspects of interfaces that go beyond the services provided by existing standards or interface areas of interest for which no standards are known. The technologies described are presented in the following categories[2]:

- Task Management Services
- User Interface Services
- Operating Systems Services
- Network Services
- Framework Services
- Data Integration Services
  - (including
    - General Administration, Commerce, and Transportation
    - Documentation
    - Electronic Design Graphics
    - Hardware Design Interface Description
    - Product Data Repository
    - Security Services
    - Software Engineering Time)

PSE Reference Model

Before it is possible to select interface standards, it is first necessary to understand all of the interface areas for which it might be beneficial to identify standards. To do this requires a thorough understanding of PSEs, both as they exist in the current state-of-the-practice and as they are expected to exist in the future timeframe of the military standard. The approach PSESWG has taken to this problem is to derive a reference model for a full environment, as no generic one existed at the initiation of the PSESWG effort. This reference model has been based on the work of the European Computer Manufacturers' Association (ECMA) and NIST, which resulted in the Reference Model for Frameworks of Software Engineering Environments[1]. These groups have contributed to the PSESWG model. Material has also been borrowed from the POSIX P1003.0 Guide to the POSIX Open System Environment[2]. The remainder of the reference model originated with the members of the PSESWG Reference Model Subgroup, making use of inputs from other organizations to the greatest extent possible.

The PSESWG Reference Model provides a catalogue of services that covers the functionality expected of a fully populated PSE. These services are divided, at a high level, into those that are part of the framework and those that are directly accessed by the end-user. The Framework Services, most of which are taken from either the NIST/ECMA reference model or the POSIX model, are categorized[3] by:

- Operating System Services
- Object Management Services
- Policy Enforcement Services
- Process Management Services
- Communication Services
- User Interface Services
- User Command Interface Services
- Network Services

---

1 All PSESWG documents described in this report are available by contacting the authors.

2 The reader will note that this list of categories is not completely compatible with the categories in the following section from the Reference Model. This was not unexpected, as the two pieces of work proceeded in parallel, and the categories will be reconciled when both documents are mature.

3 As of this writing, the reference model has not been completed. Therefore, this list is from version 0.8 and some anticipated changes for version 1.0 and is subject to change.

The End-user Services are categorized by:

Technical Engineering Services
- System Engineering Services
- Software Engineering Services
- Life-Cycle Process Engineering Services

Technical Management Services
- Configuration Management Services
- Reuse Management Service
- Metrics Services

Project Management Services
- Planning Service
- Scheduling Service
- Estimation Service
- Analysis Service
- Tracking Service
- Presentation Service

Support Services
- Common Support Services
- Publishing Service
- Presentation Preparation Service
- User Communication Services
- Administration Services

Once version 1.0 of the reference model has been published (February 1993), the next step will be to use it to identify the interfaces that are required by all of these services. That list of interfaces will be culled down to those interfaces for which standardization is likely to have a benefit. This list of target interfaces will then be prioritized and pursued for the remainder of the duration of the PSESWG.

Initial Selection of Standards

Because there are so very many PSE interfaces for which standardization might be of interest, it was felt that some sort of start needed to be made on them, despite the fact that the reference model was not yet ready. It was agreed that there are a number of basic "platform" interfaces that are provided by existing popular standards that could be easily selected without any detriment to the likely future PSESWG selections. These are standards for such things as operating system and network interfaces, which do not heavily influence the characteristics that distinguish one full PSE from another. The PSESWG only considered making these early selections in interface areas for which "obvious" standards existed, thus making the choice fairly simple.

The initial set of standards recommended for inclusion in the draft PSEI standard and the interface areas they address are:

POSIX.1 and POSIX.2 (operating system)
X-Windows (user interface protocols; the decision on a toolkit was deferred for now)
PHIGS (graphics)
GOSIP (networks)

This set of initial selections is being published as a laboratory technical report and will be widely distributed throughout the Navy and to all members of PSESWG and other interested DoD agencies. These selections are the first increment for the eventual PSESWG military standard, and they will be available for anyone's use. They are documented in "Toward a MIL_STD and MIL-HDBK for Project Support Environment Interfaces."

PLANS

The target date for the PSESWG standard is 1998. The initial selections of the whole group have been followed by the concentrated work of the Framework and Data Interfaces Teams to more thoroughly explore these two areas, coming up with ways to use the reference model to determine the interfaces that are of interest. Each of these groups is charged with:

- identifying the interface areas to be addressed
- formulating requirements for them
- identifying viable candidates for them
- and conducting an in-depth evaluation process to determine the best interface standard to select for each.

These selections will be reviewed and concurred with by the whole PSESWG and then are subject to the approval of the NGCR program office. A military standard with an accompanying military handbook will be formulated and expanded as each selection is made and approved, resulting in a formal MIL-STD and MIL-HDBK that will be submitted for formal tri-service approval late in 1996.

ISSUES

There are many issues that need to be addressed in the course of such an ambitious undertaking as the PSESWG standard. Here are a few that are likely to be of interest to anyone following this work.

Ada

The NGCR program is committed to Ada. All of the application program interfaces (APIs)[4] that are selected are expected to be provided with Ada language bindings. However, the reality of the marketplace is that there are few PSEs in existence today that rely solely on Ada interfaces of any kind, and there are even fewer bona fide standardization efforts that put a significant emphasis on Ada. Thus there are at times going to be conflicts between the desire to support Ada and the desire to adopt industry standards.

In the case of the PSESWG this apparent conflict needs to be further explored. On the one hand, the PSESWG must clearly support the functioning of tools intended to support the development and maintenance of Ada artifacts; these include, for example, compilers, linkers, program libraries, debuggers, and program design language (PDL) tools. But it is not necessarily true that all tools that support the development and maintenance of Ada artifacts must themselves be written in Ada or dependent on Ada bindings for PSE interfaces. On the other hand, examination of the marketplace indicates that the majority of work and products available (including the standards produced by accredited organizations) do not often provide Ada bindings for their interface standards.

This has put the NGCR program in an awkward position. While it is committed to Ada, it is also committed to helping the Navy become active participants in the PSE marketplace, which requires adopting industry standards that are more often in 'C'. Thus the PSESWG (and NGCR program in general) have had to accept that both Ada and 'C' language bindings will be important and will need to be supported by the interface standards that are chosen.

4 APIs are a subset of all possible interfaces: all APIs are interfaces, but there are other kinds of interfaces that do not require the procedural syntax of a programming language and so are not provided as APIs.

Community Diversity

The environments community is very diverse; it could even be said to be in a state of upheaval. Being a very young community (generally less than 15 years), there is little agreement on definition of terms. There is even a lack of consensus on the definition of what encompasses an environment or what its predominant components are (or should be). There are many relevant standardization efforts, but few have been originated by the environments community itself. There is a distinct lack of coordination, which is compounded by the fact that researchers from disciplines other than software engineering (e.g., CAD, manufacturing engineering, and concurrent engineering) are converging on similar ideas without seeming to realize the work that the software engineering community has already done. An example of the diversity is found in the use of languages, where DoD-related experimentation and implementations often make use of Ada, but industry makes extensive use of 'C' (including further divergence into new variations, such as C++), while the academic knowledge-based community often goes for LISP. In order for real progress to be made, especially in directions that will make interface standards possible and useful, such efforts need to come together and approach the problem in a coordinated, cooperative manner. It is hoped that one effect of the PSESWG reference model work will be to provide a backdrop against which such cooperation can take shape.

Profiling

Standards often have options associated with them. It is also possible that, when combining more than one standard in an effort to satisfy the needs of a whole system, there will be slight incompatibilities between some of the standards. These incompatibilities must be addressed to make the suite work together. The process of examining the individual standards in a suite with regard to their interactions and interdependencies and reconciling any differences is called *profiling*. Everyone who has ever designed and implemented a system based on the combination of two or more standards has created at least an implicit profile, but the explicit job of combining several standards and deciding the necessary reconciliations formally is a fairly new activity, and little is really known about how to do it.

The PSESWG standard will be such a profile, as it will cover many different interface areas and will cite many diverse interface standards. Thus the issue of profiling

will be a significant one for the PSE standard. There are likely to be few obvious problems with the few standards that have already been selected, as they cover rather distinct areas. But even with these there are considerations of options to include or exclude and parameter ranges to determine. And, since both POSIX and GOSIP have their own ideas about such things as time, it will be necessary to be sure that these notions can be compatibly profiled. As the number of standards in the PSESWG suite grows, addressing these compatibilities will become increasingly difficult. It may well prove to be in the best interest of PSESWG to work with other standards groups whose goals are to create similar profiles of standards that are of mutual interest.

Contentious Interface Areas

There are a few interface areas that were fairly simple for the PSESWG to address. Most of the rest will be much more difficult. There are at least three kinds of problems that can arise in trying to select one standard for each interface area:

1. Sometimes the easiest situation may be when there are two or more standards that clearly cover the same services and functionality. Then a fairly straightforward evaluation process will result in determining one to be technically and programmatically superior. This is most likely to happen in an interface area that is well-understood and for which products have existed for some time.

2. In other situations there will be several choices, but they seem to generally roam all around the same territory without being clearly comparable or, at times, clearly distinguishable. Included in this category are those situations in which there is a great deal of diversity and the marketplace has yet to establish any clear trend. This is most likely to happen in an interface area that is new but has quickly become very popular, such as has happened in the windowing arena in the last few years.

3. In yet other situations, there may be one or more choices, but none of them seems to do the whole job well. It is possible in this situation to find that two or more of the possible choices complement one another in such a way that choosing them both to support distinct parts of the interface area would be sensible. This is most likely to happen in an interface area that is being researched or slowly explored in other ways, but for which no consensus answers of what's right or works have emerged.

The PSESWG can expect to find itself in all three of these situations at one time or another in the near future. When it does, there are several options that can be considered:

- Decide: If the standards choices are there and the evaluation can be performed with satisfactory results, then making the selection will help the incremental advancement of the standard

- Defer the decision: If there is still time (as is the case right now, for example, with choosing a windowing toolkit), it may be most prudent to simply sit out and wait a while for the industry and the marketplace to sort themselves out. A lot can change in a year or two, and waiting is likely to be preferable to selecting one that turns out to be out of favor in a couple of years.

- Conduct further analysis: If things seem just too confused or overlapping, it might be best to find other ways to analyze the area or other angles to take on the requirements to make them more suitable to making a selection. Perhaps some essential distinguishing characteristic between some of the available candidates was missed in the development of the requirements. If that element can be discovered and included in the evaluation process, it may become more clear what choice to make.

- Stir the pot: In some cases, what is needed is some pro-active participation in the standards community. If things are not gelling in industry, perhaps some attempts to get important groups talking with one another or listening to the government's/user's needs will help to break the logjam and get the necessary cooperation or attention to the matter moving.

"Gaps"

Despite all of the efforts that can be made to address the selection of standards in all of the interface areas that are of interest, the PSESWG will inevitably encounter some interface areas for which there are simply no standards, despite the apparent PSESWG desires and requirements for them. In this case, some

hard choices will have to be made between at least three possible alternatives:

1. It should first be carefully considered whether or not the requirement that is not being met is real. The fact that there is no standards activity in the area may be an indication either that industry does not consider it to be a problem or they are content with current (non-standard) solutions. Re-examination of such a requirement could result in the PSESWG dropping the requirement (or at least admitting that reality suggests being willing to defer it significantly). If such re-examination confirms the importance of the requirement, then other approaches can be engaged.

2. For an important requirement that cannot be deferred, it is possible to try to drum up industry support and enthusiasm for addressing it. This is the first choice of the NGCR program, since the desire is to adopt industry standards. It may be that the PSESWG will have to put some resources (e.g., providing a working group chair or doing the paperwork necessary to get a new group or work item approved) into getting something rolling, but that would be far preferable to the only remaining alternative.

3. If by some chance the PSESWG is faced with a truly urgent requirement for which no interest can be found in industry, it may be left with no choice but to attempt to provide its own unilateral interface for the area. This could be done in parallel with trying to drum up industry support, so that the interface developed could be provided to the new group as a strawman with which they can start their work. This is the choice of absolute last resort, and it is not expected that PSESWG is likely to be forced to this.

CONCLUSION

The PSESWG is following the general tenets of the NGCR program in selecting a set of PSE interface standards. An initial selection of a few standards has been made and documented. There is no direct PSESWG experience with the application of these standards, although support environments can be found today that make use of products that conform to the selected standards or others very similar to them.

This program is consistent with a significant move throughout the industry to open systems and to the use of industry interface standards. The expected benefits include time and cost savings and increased quality, because usable products that conform to the standards will be more readily available, more technologically advanced, and more easily integrated together. The recommendations from a report[3] on the Computer-Aided Software Engineering (CASE) tool marketplace included a suggestion that it was important "to gain compatibility among platform vendors a. the environment level" and to "drive commercial standards - do not invent anything new if there is a commercial alternative." These are the objectives of the PSESWG.

The experiences of the other NGCR working groups also indicate that this program is moving in the right direction. Two other standards - SAFENET II (based largely on FDDI) for networks and the backplane standard (based largely on FutureBus+) - have been well received in the Navy and elsewhere. Even before their final formal approval they were being followed by Navy projects, indicating a wide-spread interest in the anticipated benefits and a real willingness to give this approach a try.

The PSESWG reference model has also been well-received so far. It is hoped that it will help to provide a road-map that the entire PSE industry can accept and that will help to sort out the difficulties in realizing good integrated environments that are built from a variety of vendor products.

REFERENCES

[1] *Reference Model for Frameworks of Software Engineering Environments*, NIST Special Publication 500-201. Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899; available from the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402. Also available as Technical Report TR/55, 2nd Edition, of the European Computer Manufacturers' Association (ECMA).

[2] *Draft Guide to the POSIX Open System Environment (P1003.0)*, Technical Committee on Operating Systems and Application Environments of the IEEE Computer Society, 345 East 47th Street, New York, NY 10017.

[3] *CASE Vendors' Handbook*, Dataquest, 1290 Ridder Park Drive, San Jose, CA 95131-2398. 23 July 1991.

Ms. Oberndorf graduated from Oregon State University, receiving a Bachelor of Science degree in Mathematics/Computer Science, and then attended the University of California at San Diego, receiving a Master of Science degree in computer science. She has worked for the Navy ever since, first at the Naval Ocean Systems Center in San Diego, where she helped lead the original environments work there in the mid-1970's. She led the team that developed the Common APSE (Ada Programming Support Environment) Interface Set (CAIS-A). After moving to her current location, she became involved with the NGCR program, first chairing the Operating System Standards Working Group. She now serves as the civilian co-chairman for the Project Support Environment Standards Working Group.

Dr. Schmiedekamp received his BS and PhD degrees from the University of Texas at Austin, both in Physics. He was an Assistant Professor of Computer Science at Drexel University in Philadelphia. Schmiedekamp has been a Computer Scientist at the Navy laboratory in Warminster, PA since 1984. His work has been primarily in the areas of software engineering environments and standardization. He was a member of the NGCR Operating System Standards Working Group where he was editor for a Reference Model of Real-Time Operating Systems. Currently he is a member of the NGCR Project Support Environment Standards Working Group, where he chairs the Approach Subgroup and the Standards Writing Team. His research interests include standards, reuse, metrics and multi-criteria optimization.

LCDR Squitieri graduated from the Massachusetts Institute of Technology and received a Bachelor of Science degree in Mechanical Engineering. He was commissioned an Ensign in the United States Navy in June 1980. After several tours and cruises, he attended the Naval Postgraduate School in Monterey, CA. and received a Master of Science degree in Electrical Engineering. He served his department head tour on a west coast destroyer and made a Western Pacific cruise. He is currently assigned to the Navy's Next Generation Computer Resources Program at the Space and Naval Warfare Systems Command in Washington, D.C. He serves as the military co-chairman for the Project Support Environment Standards Working Group.

NICLS: A NATURAL INTERFACE FOR A COMBINED LANGUAGE SYSTEM

John H. Gray and James W. Hooper
Computer Science Department
The University of Alabama in Huntsville
Huntsville, Alabama 35899
(205) 895 - 6515

ABSTRACT

Software systems constructed during the past two decades, and in many instances in the very recent past, have usually followed the traditional software engineering life cycle approach to software development: requirements, specification, design, code, test, and maintenance. Software developers have usually been determined to demonstrate all user requirements, but the traditional life cycle approach has often hampered this desire. Rapid prototyping has been devised to facilitate the process of software development. Since the first symposium in 1982, a variety of rapid prototyping approaches have been developed. One of these approaches involves high-level languages designed especially for prototyping. It has also become increasingly clear that discrete simulation can be used to good advantage to augment the activities in the traditional life cycle development process. As software system realizations transition over time, there is the need to achieve prototype changes easily and naturally, maintaining a consistent view of the system, and working with a language whose features may be employed in different levels of abstraction with consistency of syntax and semantics. This paper presents an overview of the development of an Ada-based interface that demonstrates the feasibility of language concepts and features that achieve a joint prototyping and simulation framework.

Introduction

Experience has revealed that customers do not always have a clear understanding of exactly what they really want their software to do. They know their application, but they cannot always describe the details of their problems to outsiders. Even if care is taken, the communication between a software engineer and the customer can lead to misunderstandings[23]. Misunderstandings can result in omitted or overlooked requirements, which will propagate throughout the software development cycle. To correct this situation, the system requirements will have to be modified, the software specification will have to be changed to reflect the new requirements, the design probably will have to be modified, the code may have to change to reflect any design

changes, and any changed code will have to be retested. These setbacks can create schedule delays. Therefore, good communication, particularly early in the software life cycle, is vitally important to the development of a functionally appropriate software system.

Sometimes newly discovered requirements conflict with previously existing requirements, thus rendering the implemented system useless. Discrepancies between what customers want and what developers provide may cost as much as 100 times more than if errors or omissions had not been made in defining the requirements during the analysis step[4]. To facilitate data flow and communication during the early steps of the traditional software life cycle, additional development techniques have been proposed. Two such techniques are discrete simulation and prototyping.

Both discrete simulation and prototyping can be used to mirror components of a "real system" relative to time passage, synchronization, communication, and the use of shared system resources, while at the same time abstracting away unnecessary detail. However, they differ in their representations of a software system. Discrete simulation produces more abstract representations of a system than prototyping, especially with respect to time passage. Discrete simulation models time passage using a simulation clock. Prototyping implements time passage using a real time clock. There is an area of uncertainty at the boundary between prototyping and discrete simulation, but the main intention of each is discernable within the software engineering life cycle[18]. In order to provide a more complete understanding of each technique, brief descriptions of both are provide below.

Discrete Simulation

As digital computers became commercially available in the 1950s and 1960s, it was recognized that they possessed a valuable capability of evaluating simulation models as discrete approximations of physical systems being studied. Because time in the physical system could be represented as a series of discrete changes, these simulation models took on the name "discrete event models", and the simulation techniques used to computerize these discrete models became known as "discrete event simulation". Over the last three decades, discrete event simulation techniques have become an integral part of some so-called "high level" programming languages. From the simulation standpoint, these languages fall into two basic categories: general-purpose programming languages and special- purpose simulation languages. The special-purpose simulation languages have developed around four discrete time control strategies: event scheduling, activity scanning, the three-phrase approach, and process interaction. The time control strategies are founded upon the fundamental concepts of conditional and unconditional events. Unconditional events are executed by sequencing events according to an agenda that is

solely time dependent. Conditional events are executed by sequencing events to an agenda that is not solely dependent on time, but is also dependent on other imposed conditions (such as, is the CPU busy?)[10].

Each discrete time control strategy determines how a modeler must view a system that is to be modeled by providing alternative world views. The term "world view" is used to describe the perspective of a system that is assumed when using a given language. Each world view has demonstrated itself as a valuable simulation approach, and in many cases has led to the development of new languages. However, the world view that has shown promise recently is the process interaction approach.

The process interaction approach has characteristics of both event scheduling and activity scanning. Any simulation language designed around the process interaction approach allows the user to concentrate on a single entity (such as a customer) and the sequence of logical steps involving the entity[1]. This sequence of steps, or activities, is commonly known as a process. Each step of a process consists of a condition segment and an action segment, which are characteristics of the activity scanning approach. The successful execution of the condition segment dictates whether the action segment is executed.

The process interaction time control procedure implements two event lists: a future event list and a current event list. The future event list, as the name implies, contains event notices for activities scheduled to be executed at some future time. The current event list, as its name implies, contains event notices for activities to be executed at the current simulation clock time, if their conditions are met. Upon simulation time update, all activities scheduled for current time are removed from the future event list and placed on the current event list. The current event list is scanned to determine if the condition segment of each entry can be satisfied. Those entries whose condition segments can be satisfied, proceed by executing the accompanying action segment. An activity progresses through as many steps as time and condition segments dictate. When an entity cannot continue to advance through the sequence of activities, the scan of the current event list can proceed and the simulation clock is advanced.

Although the process interaction strategy was accepted as one of the best frameworks for building simulation models, it was not the simplest to code. Nevertheless, this approach was implemented early in the history of programming languages via GPSS in 1961.

The current trends reflect a quantum increase in tools and methods for simulation, and all indications are that the momentum is still increasing. Due to the close conceptual conformity of a process interaction model to its corresponding system, there is every reason to expect continuing emphasis on the process

interaction world view[9].

Rapid Prototyping

It was in the decade of the 1980's that prototyping, in particular rapid prototyping, received recognition. The first symposium on the subject was held in 1982[25]. In 1983, a working conference on prototyping was held in Germany which focused on the user-oriented development of information systems. It was in the development of information systems that rapid prototyping was used effectively. Since the occurrence of these conferences, a variety of rapid prototyping approaches have developed. One technique is very-high-level languages.

Very-high-level languages are a continuation of the programming language evolutionary process. First, there was machine language which evolved into assembly language. Assembly language evolved into programming languages, which provided more simplicity and flexibility in the programming environment. Programming languages offered users the ability to potentially code several assembly language statements under the guise of a single programming language statement. This provided a more readable and logically shorter programs. Very-high-level languages take this concept one step further. Semantically related programming statements are gathered together, perhaps in the form of a procedure or function, and are represented by a single very-high-level programming language statement, thus, further reducing the logical size of a program and elevating it to a level that could be considered command driven. Very-high-level languages incorporate software reusability to provide users with a statement set that is simplistic, yet powerful.

The traditional life cycle approach did not accommodate the evolutionary development introduced by rapid prototyping capabilities[2]. However, they have been used within alternative life cycle approaches, such as the Spiral Approach. Very-high-level languages, have assumed an important position within the framework of software engineering by offering features that are uniform relative to syntax, semantics, and "system view". The intent of very-high-level language systems is to provide effective means of transitioning from lower to higher fidelity system representations, as the system realization "hardens" throughout its development[15]. Two successful examples of very-high-level language prototyping are JADE[26] and BPL[11].

Boehm, Gray, and Seewaldt[4] made a study of projects that proved to be well suited to the use of software prototypes. Their research revealed that systems constructed using prototypes performed equivalently to those systems constructed by the more traditional life cycle techniques. Even more important, their research revealed that approximately 40% fewer lines of code were written by those who used prototypes to construct their systems. As a result, rapid prototyping has proved its usefulness as a software development tool and is an area of increasing importance and research.

The Benefits of a Combination Rapid Prototyping/Discrete Simulation Language System

As can be determined from the discussion above, both discrete simulation and rapid prototyping have made a significant impact on software engineering and each has been successfully established as a useful life cycle tool. Discrete simulation has had a tremendous influence on the development of numerous computer programming languages. Rapid prototyping has had an influence by providing an alternative development technique. With these two widely used and repeatedly proven development tools available, can an advantage be gained by combining the two into a joint software development approach? If so, what are the advantages and disadvantages of the joint development approach? The answer to these questions is found by examining the advantages and disadvantages of each tool individually. Any specific advantage gained by a merger will be derived from the individual components of the union. Therefore, our attention is directed to the specific advantages and disadvantages of discrete simulation and of rapid prototyping.

Advantages and Disadvantages of Discrete Simulation

The concept of discrete simulation has been utilized for many years and the primary advantages and disadvantages are well defined. Schmidt and Taylor[24] and Banks and Carson[1] as well as others have outlined and documented them.

As a development tool, discrete simulation can be used early in the life of a software system. To assist with the analysis of such a software system, a discrete simulation model can be developed. The user can repeatedly evaluate the discrete simulation model with various data sets in order to analyze potential designs. If input data is plentiful, different proposed system designs can be evaluated in order to determine the most appropriate one.

Even when input data is somewhat sketchy, discrete simulation concepts can still be used to aid in the analysis of a proposed system. Actual system data is always preferred, since it provides realism to a simulation; however, real system data is quite often too expensive or perhaps impossible to obtain. In this case practicality is the rule of thumb. Over the years it has been shown that simulation data is usually less costly to obtain than real system data. Simulation data can be generated from a variety of existing and well proven random number generators. If system analysis can identify a probability distribution, then any necessary system data can be generated from this distribution instead of using costly data gathering techniques.

Generally, when developing a software model a software engineer can consider the model from two fundamental simulation approaches: analytical and numerical. Analytical simulation requires good insight and understanding of the mechanics of mathematics and deductive

reasoning. But being well versed in analytical methods may not be good enough. Many systems are so large and complex that mathematical models are impractical to manipulate and results are extremely difficult to deduce. On the other hand, numerical discrete simulation concepts require computational procedures which are executed on computers rather than solved manually. The executed procedures produce a system history consisting of data which can be examined to find expected results and to discover unexpected anomalies. Computers can also be used to efficiently manipulate large amounts of system data used as inputs or generated as outputs. This fact alone usually encourages many more users of numerical simulation methods over the analytical methods. In addition, analytical models usually require several simplifying assumptions to make them mathematically manageable. Discrete numerical models do not have this restriction. In many problem situations analytical models can evaluate only a limited number of system performance measures. Software engineers implementing discrete simulation models can approximate any conceived performance measure by using generated simulation data. Therefore, the ensuing cost of acquiring real data for a proposed system, or the difficulty in defining a suitable analytical model, often elevates discrete simulation as the only practical solution to the problem.

Even though simulation during software development is appealing in numerous situations, there are also some side effects from the use of simulation that are not desirable. Simulation models developed for digital computers may be costly for several reasons. Large, complex simulation models can require a good deal of time to construct and validate. The interactions that may take place within the model can be complicated and difficult to define. In order to perfect a model, numerous computer runs are required. However, this disadvantage is made tolerable through the use of the special purpose simulation languages and constantly increasing computing capability.

With the improved special purpose simulation languages and computer facilities, users can become very familiar with computing techniques, and may ignore their mathematical training. This situation can lead to bad choices for problem solutions. Sometimes users may over look an obvious mathematical solution and select a numerical solution instead. This error can prove to be costly in both time and money.

Advantages and Disadvantages of Rapid Prototyping

Rapid prototyping has been in use by software developers only about half as long as discrete simulation. Therefore, the primary advantages and disadvantages of rapid prototyping are not as well documented as those of discrete simulation. However, Connell and Shafer[7], and others provide insight into this area. We summarize these advantages and disadvantages in the following paragraphs.

As a communication tool, rapid prototyping has demonstrated itself to be helpful and sometimes cost effective in many small and large projects. Consider the situation involving an incomplete requirements document. It is very difficult to achieve coordination between the design specification and all user requirements for a complex system until the user has witnessed at the minimum a partial implementation. When a customer attempts to relate requirements to a system engineer, uncertainty exists about what is needed in the software. Users are busy, they forget important details, or they really just aren't interested. Sometimes, not everyone who should be involved in requirements definition is involved and very important criteria is lost[5]. In this case the software requirements may be considered a "wish list", but not necessarily a complete wish list[23]. At other times, the customer may have a basic understanding of what is needed in the desired software, but is uncertain if it is feasible to develop it. With rapid prototyping, sections of any proposed system can be represented and used to determine the feasibility or desirability of the customer's perceived requirements. The requirements can be evaluated before the design is finalized. Formal specifications may never be documented as in the traditional life cycle approach; however, if specifications are desired, the final prototype itself may be perceived as the specifications for the software[22].

The experience and understanding gained in constructing a prototype transfers into an overall better system. In terms of development cost and development time, it appears to be more effective if changes to system ambiguities occur early in the software development, rather than later during the maintenance phase after the software has been released. Historically, the functionality of a system is changed by users late in the development cycle because software engineers usually do not greatly involve users during the pre-design phases. As a result, a user's perspective sometimes turns out to be different than the design specification in the reality of a visible, working system.

However, using rapid prototyping, the opportunity to make functional modifications as a result of changes in the requirements can come at a much earlier time. Users have an opportunity to view a working model prior to test and maintenance. Therefore, an advantage of rapid prototyping is accommodating new or unexpected customer requirements early in the development cycle rather than later during the test and/or maintenance stages.

A direct consequence of involving the user early in the software development process is the cost savings that can occur in the maintenance phase of software development. Traditionally, using the software development life cycle, most of the cost of software is not incurred during the development effort or during the coding effort, but rather during

software maintenance[7]. It is a fact that software engineers make development mistakes during the early phases of the software life cycle. Some of these errors surface during the test phase, and others will go undetected until after the software is operational. In the traditional life cycle approach a great portion of the development effort is directed to correcting mistakes that propagate through the early phases of software development into software test and maintenance[7]. By using rapid prototyping, cost reductions in the test and maintenance phases occur because software engineering errors have a greater opportunity to be caught by the users before the test phase begins. Therefore, fewer modifications will be required after software delivery.

An additional advantage gained from rapid prototyping is the ability to discard the prototype of an infeasible or unworkable system at a relatively small loss compared to the devastating loss from discarding a system late during the test or maintenance phase of the classical software life cycle. A software system that survives the inspection of the user through a rapid prototype can only increase the user's satisfaction with the final product, thus providing a measure of quality control and productivity improvements in the development phases beyond the analysis prototype design.

One disadvantage with using rapid prototyping is not found in the technical approach implemented by the users, but rather in the way users perceive it. Since rapid prototyping is a relatively new technique for software development, not all software engineers feel comfortable with it. Although a software system may be developed by using rapid prototyping techniques, maintenance engineers or even software designers may be more accustomed to the traditional life cycle approach. A change in approach may be perceived as criticism of their professional ability and they may become uncomfortable with the situation or possibly even hostile. There is always an inclination to reject techniques that are not familiar, and thus delay their advancement.

In the same vein there is a shortage of trained prototypers and maintainers of prototype systems. Compounding the problem, there has been little opportunity for training prototypers or introducing developers to this newer way of thinking. Companies that develop prototyping tools usually offer training, but a general approach to rapid prototyping and rapid prototyping methodologies has been lacking.

A Merger of Rapid Prototyping and Discrete Simulation

With rapid prototyping and discrete simulation working together in a combined language system, benefits emerge in all aspects of the software life cycle. To better understand how these benefits relate to the traditional software cycle, consider the software life cycle to consist of the following six phases:

(1) requirements analysis
(2) system design

(3) high-level software design
(4) implementation
(5) integration/testing
(6) operation/maintenance

In phase one, during the early stages of the software development, it is very prudent to determine the feasibility and practicality of individual requirements. To assist in this operation, a simulation can be developed that can examine software throughput data and determine if the volume of data defined in a requirement is reasonable. If a sufficient amount of real data is not available to the simulation, random number generators can be used to supply any amount of data necessary to test these requirements. It is also valuable to inspect some potential man/machine interface requirements at this point. A rapid prototype can be used to accomplish this task. Using a very low fidelity representation of the software's functionality, plausible man/machine interfaces can be constructed and examined for their usefulness to potential users[18].

During phase two, system design, assessment capability is essential to proper software development. Assessment of resource utilization and queue buildup is vital to the success of phase two. Efficient use of system resources and data structures is critical to the development of a usable baseline. Simulation techniques can be used to perform these assessments.

In phase three, high-level software design, proper assessment can reveal input/output problems, as well as identify high activity modules that require special attention from the developer. In either case simulation activities can again be used to expose software flaws in order to avoid potential bottlenecks during software operation. Phase three also provides an excellent opportunity for usage of prototyping techniques, to resolve issues concerning distributed processing capabilities.

After the software system transforms from software design into an implementation, prototyping and simulation can still be valuable tools. Software engineers can build prototype modules to serve as drivers and stubs during integration and testing (phase five). In phase six, simulation and prototyping can be used to evaluate the impact of changes to working software prior to any actual change.

To some engineers the concepts of rapid prototyping and discrete simulation are very similar and may cause some confusion. There is a "gray area" at the boundary between rapid prototyping and discrete simulation, but the main purpose of each development approach is distinguishable. Simulation clearly has its most effective role in the more abstract representations of a software system during requirements analysis, design assessments, and maintenance. Prototyping has its role in the more visible aspects of the software that occur during system design, software design, and integration and testing. As the software transitions through the software life cycle, and the representations of system

characteristics become more concrete, it may be more appropriate to use "real" time instead of simulation time. This change implies a natural flow from simulation to prototyping crossing each phase, yet maintaining a consistent view of the software system under development. By using a joint simulation/prototyping language system, this time flow capability can be achieved[17,18].

NICLS

To demonstrate the validity of a combined language system supporting discrete simulation within a rapid prototyping environment, a study has been conducted with emphasis on developing an interface to discrete simulation languages and rapid prototyping languages. The interface was named NICLS (Natural Interface for a Combined Language System) and is pronounced "Nichlas". The purpose of the study is fourfold:

(1) to select a discrete simulation language, which possesses a set of language statements that will serve as a representative of the discrete simulation languages,

(2) to select a rapid prototyping language, which possesses a set of language statements that will serve as a representative of the rapid prototyping languages,

(3) to utilize the programming language features of the discrete simulation representative and the rapid prototyping representative by means of an interface from a general purpose programming language,

(4) to select a general purpose programming language to serve as a base language as well as a medium in which the interface can be developed.

The results of the study yielded the selection of Ada as the general purpose programming language, the Behavior Prototyping Language (BPL)[11,12] as the representative of rapid prototyping languages, and Simulation and Modelling on Ada (SAMOA)[21] as the representative of discrete simulation languages. The basis for each of these selections is outlined below.

Ada as a General Purpose Programming Language

Although relatively new as a general purpose programming language, Ada has obtained a respected following within the software engineering community. It was developed under the leadership of the United States Department of Defense to be used initially within large, real time embedded computer systems. Ada proved to be adaptable across a diversified set of applications in both the commercial and the government communities. Ada's success was due to its suitability across the software life cycle. It possesses features designed to facilitate both classical software engineering and rapid prototyping[8]. Some of these features are the more modern software engineering principles: data abstraction, information hiding, modularity, concurrency, portability, strong typing, and versatile syntax. These principles allow Ada to bridge the gap between past programming languages and current software

development methodologies.

BPL

BPL was selected as the representative of the rapid prototyping languages based on the following criteria: language features, proven application, and availability of information. BPL offered an excellent example of a very-high-level rapid prototyping language, and it provided stable guidance in the development of the rapid prototyping features.

BPL was designed to provide language features uniform relative to syntax, semantics and system view. It was purposely designed to achieve rapid prototyping of system behavior by use of abstraction, which can be performed to different levels with good effect. The syntax and semantics of BPL language statements were influenced by several existing languages, the most influential being SETL[19], Simscript[20], and Pascal. BPL is not a "strong data typing" language due to the influence of set-theoretical concepts from SETL. Rather BPL is a strong object-typing language.

BPL was designed by Dr. James W. Hooper. It was based on research performed at the University of Alabama in Huntsville. It was developed for the United States Army Ballistic Missile Defense (BMD) Advanced Technology Center for use on their Distributed Data Processing Testbed. Dr. Hooper is serving as advisor to the NICLS research effort and is available to provide first-hand information concerning the implementation of BPL.

SAMOA

SAMOA was selected as the representative for the discrete simulation languages based on two criteria: base language and simulation world view.

SAMOA, as the name implies, has Ada as the base language and was an attempt by its developers to apply Ada constructs toward the development of a fully integrated, discrete event simulation language. The design of SAMOA was guided by two goals:

(1) limiting the number of Ada programming statements a user must know to a Pascal equivalent subset. This could potentially ease the transition of non-Ada programmers to an Ada-based language, and

(2) allowing users, when necessary, to have access to the full power of Ada. This provides the knowledgeable Ada programmer a full arsenal of Ada language capabilities to apply to the problem at hand.

By possessing a Pascal like subset, SAMOA provides a rich supply of known programming features to integrate with the BPL programming features.

SAMOA approaches discrete simulation from the process interaction world view, and demonstrates a continued trend of increased usage of this world view in discrete simulation languages within the United States.

The Guiding Principles in the Development of NICLS

The guiding principles used

during the development of NICLS follow the principles outlined in Hooper[13]. Those principles are: understandable language concepts, understandable language syntax, computational/representational power and flexibility, and support for the software development process. Although the principles discussed by Hooper were simulation oriented, their applicability to NICLS is just as valid.

The Overall Approach to NICLS

NICLS provides a syntactic/semantic approach to software development. All NICLS statements are syntactically identical whether they appear in a simulation or in a rapid prototype. However, their interpretation depends on the semantics of the module that contains them. A preprocessor is used to translate the NICLS statements into standard Ada.

A program may contain a combination of NICLS and Ada statements. The preprocessor scans for the NICLS statements while ignoring the accompanying Ada statements. It recognizes the NICLS statements because they have an augmented Ada-like syntax that is distinguishable from standard Ada by the preprocesor. Once the preprocessor has completed the translation process, the Ada compiler provides the run- time support.

The Combined Language Features of NICLS

Within any software development effort there exist aspects that can be considered the principal aspects of the system. In many instances these aspects can be defined using abstract representations. However, unless there exists some set of implementable mechanisms corresponding to the abstract representations, these abstractions will be useless. In order for NICLS to be a viable software development tool, it must demonstrate a useful set of implementable mechanisms. Thus there remained the task of identifying for NICLS these implementable mechanisms.

Some of the mechanisms can be easily identified. For example, consider "Time Passage" as the candidate for abstract representation. Time passage from a software developer's perspective can be envisioned as two distinct clocking mechanisms, simulation time and real time. If simulation time is selected as the desired clock strategy, then the corresponding mechanism can be a transformation module that operates in a simulation framework (i.e., a simulation clock and a future event list). If real time is chosen as the desired clock strategy, then the corresponding mechanism is a transformation module that operates in a prototyping framework (i.e., real time delays).

By using the time based transformation modules described above, an added bonus is revealed. If the transformation modules are implemented so that the programming statements composing them are identical syntactically, and the only difference between the simulation transformation module and the prototyping transformation module is the interpretation applied to the programming statements, then

a key programming element that is necessary for a clean transition from low fidelity to high fidelity is manifested. This key programming element is consistency. Consistency occurs because the transformation modules look the same from a programming perspective, but the interpretation is dependent upon the semantics implied by the implementation. This establishes a working framework that can evolve semantically from a simulation framework to a prototyping framework.

To properly use this key programming element to its full potential requires that the programming framework contain transitional statements that appear syntactically identical, yet are translated according to the semantics implied by the framework containing them. An example of such a transitional statement is the DELAY N statement. In a simulation framework DELAY N causes some event within the simulation to be placed on the future event list. When the simulation clock is updated by N time units, the event is removed from the future event list and reactivated. Alternatively, DELAY N in a real time framework causes some event to become idle until N real clock units have elapsed, then the event is reactivated. Notice that the statement, DELAY N, has the same syntax for both implementations, but the interpretation is dependent upon the semantics of the enclosing framework.

In order for programming statements like DELAY N to have meaning as transitional statements within NICLS, they must be viable programming statements in both simulation languages and prototyping languages. This concept is currently directing the investigation leading to the definition of a set of transitional statements. A comparison for syntactic uniformity is being made using programming statements from both SAMOA and BPL. It is our goal to complete this study and have a working version of NICLS by the end of this year.

REFERENCES

1. Banks, Jerry and Carson, John S. 1984. Discrete-Event System Simulation, Prentice-Hall, Inc. Englewood Cliffs, N. J.

2. Belz, Frank C. 1986. "Applying The Spiral Model: Observations On Developing System Software In Ada", in the Proceedings of the 4th Annual National Conference on Ada Technology. (Atlanta, Ga. Mar 19-20, 1986). U. S. Army Communications-Electronics Command, N.J., 57-66.

3. Boehm, Barry W. 1983. Software Engineering Economics, Prentice-Hall, Englewood Cliffs, N.J.

4. Boehm, Barry W., Terence E. Gray, Thomas Seewaldt, 1984. "Prototyping Versus Specifying: A Multiproject Experiment", IEEE Transactions on Software Engineering SE-10, no. 3(May): 290-302.

5. Buhr, R., J., A., 1984. System Design In Ada, Prentice Hall, Inc., Englewood Cliffs, N.J.

6. Carrio, Miguel A. 1986. "The Technology Life Cycle And Ada", in the Proceedings of the 4th Annual National Conference of Ada Technology. Atlanta, Ga. Mar 19-20, 1986). U. S. Army Communications-Electronics Command, N.J., 75-82.

7. Connell, J. L. and Shafer, L. B. 1989. Structured Rapid Prototyping, An Evolutionary Approach to Software Development, Prentice Hall, Englewood Cliffs, N.J.

8. Duncan, Arthur G. 1982. "Prototyping in Ada: A Case Study", ACM Sigsoft Software Engineering Notes 7, no. 5(Dec): 54-60.

9. Gray, John H., James W. Hooper, 1987. "The Evolution Of Process Oriented Simulation Languages", in the Proceedings of the 1987 Southeastern Computer Simulation Conference. (Huntsville, Alabama, Oct 19-21, 1987). SCSC, San Diego, Ca.

10. Hooper, J.W. 1979. Simulation Strategies: A Theoretical and Practical Investigation, A Dissertation, University of Alabama, Birmingham.

11. Hooper, J.W. 1985a. "BPL: A Set-Based Language for Distributed System Prototyping.", International Journal of Computer and Information Sciences 14, no. 2(Apr): 83-105.

12. Hooper, J.W., J. T. Ellis, T. A. Johnson, 1985b. "Distributed Software Prototyping with ADS", in the Proceedings of the 8th International Conference on Software Engineering. (London, U. K., Aug 28-30), IEE, Los Angeles, Ca. 216-223.

13. Hooper J. W. 1986. "Language Assessment Criteria For Discrete Simulation", in the Proceedings of the 1986 Winter Simulation Conference, 404-408.

14. Hooper, J. W. 1987a. "Software Engineering Approaches For Discrete Simulation", in the Proceedings of the 18th Annual Conference on Modelling and Simulation, (Pittsburg, Pa., Apr 1987).

15. Hooper, J. W. 1987b. "Uniform Language Concepts for Prototyping and Simulation of Software." in the Proceedings of the 1987 Summer Computer Simulation Conference (Montreal, Quebec, Canada, July). SCS, San Diego, Ca., 93-98.

16. Hooper, J.W. 1987c. "Languages Features For Discrete Simulation", Computer Languages 12, no. 1: 39-46.

17. Hooper, J. W. 1988a. "Programming Languages." System Modeling, N.A. Kheir, ed. Marcel Dekker, Inc., New York, N.Y., 291-317.

18. Hooper, J. W. 1988b. "Language Features For Prototyping and Simulation Support Of The Software Life Cycle", Computer Languages.

19. Kennedy, K., and J. Schwartz 1975. "An Introduction to the Set Theoretical Language SETL", Comp. and Math. with Applications 1(1): 97-119.

20. Kiviat, P. J., R. Villanueva, and H. M. Markowitz 1973. SIMSCRIPT II.5 Programming Language, 2nd Ed., Consolidated Analysis Centers, Inc., Los Angeles, Ca.

21. Lomow, G., B. Unger. 1982. "The Process View of Simulation in Ada." in the Proceedings of the 1982 Winter Simulation Conference (San Diego, Ca., Dec 6-8). IEEE, N. J., 77-86.

22. McCracken, Daniel D., Mickael A. Jackson 1982. "Life Cycle Concept Considered Harmful", Software Engineering Notes 7, no. 2(Apr): 29-32.

23. Pfleeger, Shari Lawrence, 1987. Software Engineering. Macmillan Publishing Company, New York, N.Y.

24. Schmidt, J. W., and R. E. Taylor 1970. Simulation and Analysis of Industrial Systems, Irwin, Homewood, Ill.

25. Squires, S.L., M. Branstad and M. Zelkowitz (quest editors). 1982. "Special Issue on Rapid Prototyping", ACM Software Engineering Notes, 7, no. 5(Dec), 1-185.

26. Unger, B., G. Birtwistle, J. Cleary, D. Hill, G. Lomow, R. Neal, M. Peterson, I. Witten, B. Wyvill, 1984. "JADE: A Simulation and Software Prototyping Environment", in Simulation in Strongly Typed Languages: Ada, Pascal, Simula ..., Society for Computer Simulation, San Diego, Ca., 77-83.

Brief Bibliography

JOHN H. GRAY is employed by TF Inc. as a senior softwa engineer working on the Spa Station Freedom project. He currently pursuing a Ph.D. Computer Science at t University of Alabama Huntsville.

Dr. JAMES W. HOOPER is current serving as Weisberg Professor Software Engineering at Marsha University, Huntington, W.V., leave from his position Professor of Computer Science the University of Alabama Huntsville. Earlier he conduct simulation research as employee of NASA Marshall Sp Flight Center.

STRAda: A software tool for distributed Ada

D.Bekele, C.Bernon, M.Filali, J.M Rigaud, A.Sayah

IRIT

Université Paul Sabatier

118, Route de Narbonne

31062 TOULOUSE Cedex

email: bekele@irit.fr

Abstract

Ada was designed for programming in the small as well as for programming in the large. As far as programming in the large is concerned, its main features are:

- packages for modularity,
- tasks for concurrency,
- and exceptions for reliability.

Currently, these features are well supported for centralized environments, however they raise some problems in distributed environments. Several ways of addressing the issue have been considered: for instance, in Ada 9X new programming language constructs, e.g. virtual nodes, are proposed; we investigate an alternative approach which consists of a software tool for transforming standard Ada programs into distributed Ada programs.

1. Introduction

The importance of distributed systems has undergone constant growth over the past decade. This trend is due to a number of factors including:

- the decreasing cost of processors,

- the continuously increasing power of these processors,

- the high capacities and speeds achieved by computer networks.

On the other hand, the complexity and the cost of applications running on these powerful systems have taken on considerable proportions[9]. The main reason lies in the fact that it is often difficult to use software design concepts and tools which are mostly intended for non-distributed programming. High level languages very rarely include abstractions which make distribution easy to handle. As a result, information characterizing the hardware configuration of the network is directly embedded in the application code. Structures of this kind are a serious obstacle to the portability and maintainability of the application.

It is also true that programmers may not feel that the task of distributing their applications actually concerns them. The use of several processors is a simple way of accelerating the execution speed of their programs. in this case, the description of distribution functions by the programmer becomes a painstaking task. The problem is now one of implementation and optimization of the application.

Ada is currently one of the few languages possessing a fully integrated parallelism model. However, version Ada83 incorporates no specific distribution features. More particularly, the language does not define the unit of distribution (task, package, etc.) and contains no abstraction suited to the problem (processor assignement for tasks, inter-process communication and synchronization, etc).

The STRAda system (*Système de Transformation et de Répartition Ada* or *Ada transformation and distribution system*) completes Ada and provides a simple way of solving the distribution issue. It is based on the following principles:

- The task, which is the unit of parallelism in Ada, is chosen as the distribution unit of the application;

- Processor assignement for tasks at the various network nodes is performed either statically (programmed mode) or dynamically (interactive or automatic mode);

- Inter-node communication is entirely handled by the system and remains transparent to the programmer.

The STRAda system consists in translating one Ada program into another Ada program. In the transformed program, tasks are replaced by distribution units and inter-task communications become inter-node communication. The system provides of a user interface used to place, monitor and control tasks in operational mode.

In part one of this article the authors classify projects dealing with the Ada distribution issue. Parts two and three specify and substantiate the choices adopted in the STRAda project. Implementation is dealt with in part four. The last section provides an evaluation of completed work.

2. Current approaches of addressing the issue

Although the Ada language has not excluded the distributed aspect of applications , it does not include the basic concepts necessary to a simple expression of this distribution, as is the case for parallelism and other advanced software engineering concepts. A number of investigations have been brought to bear on this aspect of programming which address the issue from various different angles. A classification system developed by Bishop and Hasling [8] sums up these methods:

1 - How many Ada programs are involved?

Either a single Ada program is partitioned for distribution over the network or several Ada programs are run locally at each node and communicate with each other across an independent message system.

2 - How is communication among nodes expressed?

Communication can be explicitly expressed by the programmer of the application, or it can be implicit and transparent to the programmer when generated by the translator or tool dedicated to this service.

3 - What is the degree of liberty left to the programmer as far as the unit of distribution is concerned?

A number of language constructs could be considered to comply with the profile of a distribution unit, the most important being packages and tasks, although subroutines and even variables could also be distributed.

4 - How is the distribution map specified?

Distribution of units to the various nodes can be explicitly described, using a dedicated language for instance, or automatically by a dynamic topographical analysis of the network used.

The authors define several ways of addressing the problem according to the choices made with respect to these criteria. The disadvantage of the most fundamental approach (several Ada programs)[29] may be that checking among programs running on several nodes could fail. However, this pitfall could be avoided by using dedicated communication packages.

The next two classes involve single Ada programs and imply constraints on distribution units. Communication can either be explicit [14, 1] or implicit [27, 2, 10].

The last class explored involves Ada applications in which no constraints are made on distribution units. The most advanced project in this area is the APPL project [15, 13].

3. STRAda and Software Engineering

The STRAda system consists in transforming a classic Ada program into another equivalent program. The essential aim of the transformation operation is to take into account the existence of several processing nodes in order to globally increase the performance of the application. The extensions fully comply with the initial semantics of the language and are based on the most fundamental software engineering principles.

3.1. Portability

Portability is the ease with which a software product can be adapted to different hardware and software environments [22]. Portability is improved by applying software engineering principles such as abstraction and data dissimulation.

The basic principle of abstraction consists in extracting the vital property of one level while omitting details that are not crucial [24]. A high level abstraction specifies **which** actions must be performed while a lower level abstraction specifies **how** each action is to be completed

The underlying principle of data dissimulation consists in rendering inaccessible certain details which must not affect other parts of the system. In STRAda, for instance, the protocol used on the network is hidden: authorizing it would constitute a breach of distribution abstraction logic.

Ada is one of the rare widely used languages to incorporate a parallelism model. The Ada task is an abstract model of parallelism and is independent of the host system. The rendezvous concept lets tasks synchronize and communicate. This property is essential in designing portable parallel applications.

On the other hand however, Ada does not provide an abstraction which can be used to express distribution. In the case of distributed applications, the programmer must therefore use the distribution model provided by the host system. In UNIX systems for instance, the distribution model is defined by the process as a distribution unit and sockets the means of communication between processes.

Consequently, the portability of such applications is badly compromised by the fact that implementation details which are too close to the hardware used are taken into account.

The STRAda system used the Ada parallelism model as the basis of the distribution model. Indeed, parallelism and distribution have many points in common. Using the same unit for both parallelism and distribution is therefore natural, the same being true for communication and synchronization mechanisms. This association enables the STRAda system to render distribution totally transparent to the programmer. It can be said that parallelism in Ada is a programming concept which is available to the programmer and that distribution with STRAda is a programming concept which is hidden from the programmer.

3.2. Reliability

Reliability is the ability of software to operate even under adverse conditions [22].

One of the most fundamental requirements of embeded systems, for which Ada was specifically designed, is reliability. This means that no errors are allowed to remain in the system. Residual errors are nevertheless inevitable such as erroneous input data, for example. When exceptions of this sort occur, such as a divide by zero, most languages stop processing

and the operating system takes over: this should not be true of a reliable real-time system.

Ada exceptions allow detecting abnormal situations and processing them with specific action. Detection of abnormal situations is ensured by the hardware, system software or the user application. Any exceptions detected must compulsorily be processed.

In distributed Ada applications, the exception concept cannot be applied to the whole application. In effect, either the application consists of several independent applications, in which case exceptions vary from node to node, or the application is a single program, in which case inter-node communication generally does not cater for exception transmission. This type of situation occurs when exceptions take place during a rendezvous between two tasks located on different nodes.

3.3. Reusability

Reusability is the ability of software to be totally or partially reused for new applications [9]. Many software components are similar: it would be interesting to exploit these similarities to avoid performing redundant work. In distributed applications, there is often a need for a program which already exists for a different hardware configuration (different number of processors, different processor types, etc.)

Reusability directly affects all other cost and quality factors. When all or part of an existing software program is reused, development costs are diminished in all the phases of the software life cycle. Software components already tested on other target systems are used, which increases the reliability of the application.

Ada provides a high degree of reusability thanks to features such as genericity: i.e. the ability to parametrize a library unit. Parametrization increases the reuse potential of these units.

Since no distribution units exist in Ada, data relating to the hardware configuration the application is actually going to run on are disseminated throughout the program. Reuse of the program for another configuration (also called reconfiguration) is not a simple task since all levels of the distributed application must be reworked.

With STRAda on the other hand, all information concerning the hardware configuration are supplied during another phase which is totally separated from the application logic programming stage. Reconfiguration only involves modifying this phase dedicated to configuration: the Ada program pertaining to application logic is not modified.

3.4. Other characteristics

Efficiency means making good use of available hardware resources [22]. It has often been thought that the efficiency of an application diminishes when the level of the language used to program it increases. The contrary has been proved with applications written in Ada and rewritten in assembly language. Indeed, the executable originating from Ada is sometimes extremely efficient. This can be explained by the fact that the compiler uses several optimizing techniques that assembly programmers do not use.

In the same way, optimizing techniques can be implemented (processor assignement to balance out loads, use of the right protocol, etc) in the STRAda system which can make applications using this system more efficient.

Ease of use refers to the ease with which the users of a software program learn to operate it [22]. It is obvious that the higher the abstraction level, the greater the ease of use. As STRAda possesses a higher level of abstraction for distribution than distributed programs written in Ada83, it is easier to use.

4. STRAda and the solutions adopted

In the previous sections we have discussed the importance of a high level abstraction model for distribution. In the STRAda project, we have reused the Ada task model for distribution. This choice is in line with the target architecture: a network of workstations. A finer degree of distribution, such as that adopted by APPL[15] seems to us to be better suited to a massively parallel architecture (example : a transputer network).

The choice of using Ada tasks as the distribution units and the desire to maintain Ada concepts such as exceptions for the distributed program lead us to adopt the single program model for the distributed application.

The basic principle of the project consists in developing a minimal distribution kernel used to implement all the more specific concepts of the Ada language. For the minimal kernel, we have opted for communication via message exchange. A certain number of aspects of the language are not accounted for by the distribution kernel such as shared variables and task termination. In section 5 we shall discuss

implementation of these points in terms of services provided by the minimal kernel.

Implementation consists in translating an Ada source program into another Ada program (figure 1). In the transformed program, all constructs relating to parallelism and communication are replaced by calls to services provided by the STRAda kernel.

To conclude, some of the most interesting facets of STRAda include a minimal kernel adapted to the Ada language and reuse of existing systems without any modification (Ada compilers, UNIX operating system etc.).

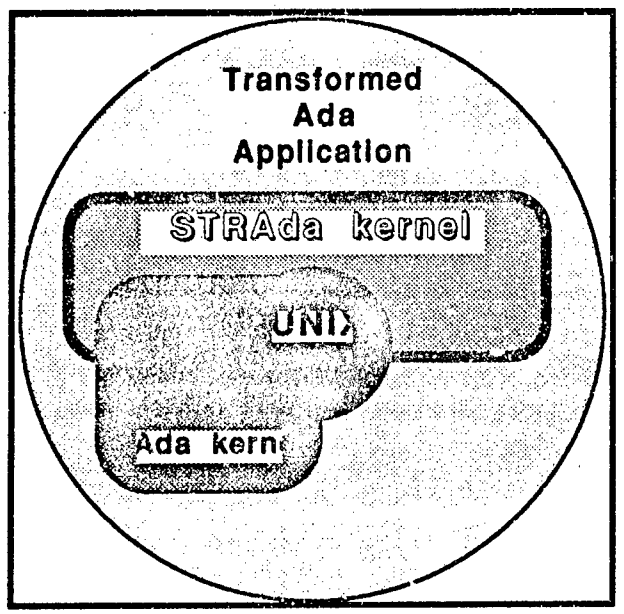


Figure 1: The STRAda kernel

5. Defining a distributed kernel adapted to Ada

In this section, we shall look at the relationships we have identified between the concepts and mechanisms of the Ada programming language and the UNIX operating system. We shall do this by examining parallelism, synchronization and distribution issues.

5.1. Parallelism

We have quite naturally associated a UNIX process with each Ada task. However, from a purely practical point of view, this may seem costly: we could have considered using the technique found in most compilers of multiplexing several tasks within a single UNIX process (pseudo-parallelism). Nevertheless, the implementation strategy adopted has the following advantages:

- real parallelism within an Ada program is made possible on multi-processor UNIX stations;
- the underlying operating system (UNIX) can be reused more efficiently, as the synchronization of an Ada task is expressed directly in terms of the synchronization of the support process.

In the STRAda system, parallelism and distribution are implemented by remote task creation. The site where the task is resident is currently specified explicitly when the task is created.

5.2. Synchronization and communication

In order to install the distributed version of Ada's synchronization and communication instructions, we have chosen sockets to support synchronization and communication. Each Ada entry is associated with a socket and a port. The UNIX primitives (*sendto*, *recvfrom*, *select*) make it possible to send or receive on one of these entries or to select an entry from a set of entries.

Currently, each Ada task handles its own rendez-vous; we could have designed a more efficient strategy with each site having a single rendez-vous server.

The solution adopted has certain limitations, but it also has the considerable advantage of enabling direct reuse of the UNIX communication system.

5.3. Exceptions

We have seen that developing a distributed application with STRAda frees programmers from the need to plan for distribution. This rule still applies where an exception is raised at a site and has to be handled on another site.

In a non-distributed application, the Ada run time system handles the propagation of the exception and locates the corresponding exception handler. In a distributed application with STRAda, the STRAda kernel handles the propagation of the exception between two remote sites where necessary. As the distribution unit is the task, such propagation is required in the following cases:

- when an exception is raised during a rendez-vous;

- when attempting to rendez-vous with an aborted task;

when an exception is raised as a task is being created.

It is useful to be able to handle exceptional circumstances that can arise in distributed systems (such as communication or processor failures) by using exceptions. The STRAda kernel must therefore be capable of raising such exceptions and propagating them to the destination tasks.

An exception that has to be propagated from one distributed task to another must be intercepted and sent to the remote program in coded form. When the program receives it, it is then decoded and the original exception is explicitly raised in the destination task. The Ada kernel then takes over on site.

5.4. Architecture and implementation of the STRAda kernel

The STRAda kernel consists of:

- a package providing tasks with services for parallelism, communication and synchronization;

- a set of tasks, called Server-Creation, each executing at each application site (see figure between: STRAda Architecture).

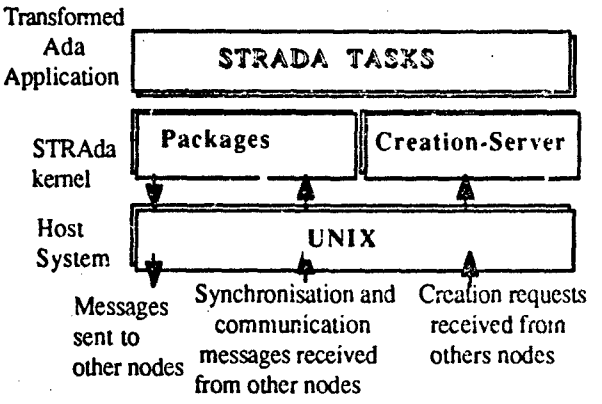


Figure 2: STRAda Architecture

An application task synchronizes itself or communicates by using Ada's dedicated instructions (i.e. task declaration or dynamic task creation, entry call or acceptance of a rendez-vous on an entry, *select*, etc.). These subroutines use sockets to communicate directly with a remote task, or with a Server-Creation task. Each Server-Creation task initializes tasks to be executed at the site. It does this by creating a UNIX process and assigning it a unique name. In this way, to create a task on a site S, the parent task communicates via the STRAda package with the site's Server-Creation task, which then returns the data that the parent task needs to synchronize or communicate directly with the task created (entry points represented by UNIX sockets). The table below shows the contents of the package in summary form.

STRAda primitives	Function performed
CREATE	Create task
CALL	Call task entry
ACCEPTT	Accept rendez-vous without exchange of data (caller not blocked)
ACCEPT_DO	Accept rendez-vous with exchange of data (caller blocked until end of rendez-vous)
SELECTT	Accept one of several rendez-vous, immediate or deferred

6. Transforming the Ada model into a UNIX model

In this section, we shall take a brief look at how the transformations proposed are implemented. We start off with an overview of the tool we used, the Cornell Synthesizer Generator [26] (for more details and examples, refer to [25]). We then go on to discuss the transformation principles and choices adopted.

6.1. The Cornell Synthesizer Generator

The Cornell Synthesizer Generator (CSG) is a tool that uses an assigned abstract syntax to generate a syntax editor. The programming language used by the CSG is the Synthesizer Specification Language (SSL). In SSL, abstract trees are specified using the concept of a phylum. A phylum is a type of tree with nodes that are constructed recursively using specific operators and phyla. For the STRAda project, we chose the abstract syntax DIANA [12] (Descriptive Intermediate Attributed Notation for Ada). The main reason for this is that DIANA is considered to be the standard notation for Ada programs, so STRAda will be able to integrate with other tools like Anna [18]. Another important feature of the CSG is that the abstract trees may contain synthesized or inherited attributes [11]. These attributes are normally used to express the language's semantic checks. We have used them in the STRAda project to synthesize other programs.

6.2. Transformation principles

Here, we shall look at the transformation principles used in an Ada program. We should recall that the aim of transformations is to replace all Ada instructions concerning parallelism and synchronization with equivalent instructions calling the STRAda kernel we have already described. We concentrated mainly on:

- a global functional transformation in which the abstract tree of a program is completely transformed by an SSL function;
- a local attribute transformation in which each node of the abstract tree of an Ada program synthesizes a transformed node, the global transformation therefore consisting of the attribute of the root of the program's abstract tree.

The main point to stress is that whatever type of transformation is used, a new Ada program is obtained.

6.2.1. Functional transformation

As we have already seen in the previous section, the CSG's SSL language is capable of defining and handling abstract trees of an abstract syntax. Here, we want to transform abstract trees in DIANA notation. This transformation can be defined as a recursive function on the abstract tree of a program: for each type of node *N*, we define a transformation function that defines *N* and, if necessary, recursively invokes the functions associated with the types of the child nodes of *N*.

Although theoretically possible, for the moment this strategy can only be implemented using the CSG. What normally happens is that a transformation function references attributes associated with a node in the abstract tree, which may be the environment inherited from this node, for example. However, referencing an inherited or synthesized attribute in an SSL function is not possible using the current version of the CSG.

6.2.2. Transformation by calculation of attributes

In attribute transformation, we only define the local equations required to relate each node to an attribute representing its transformation. Provided that circularity is avoided, an attribute definition can reference other attributes.

This is the strategy that we have adopted. An interesting feature is that the abstract tree of the initial Ada program is not modified, as the Ada program of the application to be transformed is the attribute associated with the root of the original Ada tree.

6.2.3. Transformation examples

The examples below are designed to show how an Ada program is transformed into an equivalent Ada program by going through the STRAda kernel. The first example is a simple one based on the *accept* instruction only, and the second example shows how the select instruction is transformed.

Example 1

```
type MSG is array (1..n) of character;
accept RETRIEVE (INFO : MSG) do
   INFO := BUFFER;
end RETRIEVE;
```

This Ada code is translated into the equivalent Ada code:

```
declare
   Info : msg;
begin
   Accept_do(Task_Box.entry_RetrIEve,
Caller_name);
   Info := BUFFER;
   ACCEPT_END
(TASK_Box.entry_Retrieve,
      Tp_out => (Tab_param'first =>
(Info'address, Info'size),
      Caller_name);
end;
```

Example 2

The UNIX *select* primitive, which waits for a message on a set of ports, enables the Ada *select* instruction to be easily translated into the equivalent Ada code:

```
select
   when S>0 =>
      accept P:
      S := S - 1;
   or
      accept V;
      S := S + 1;
end select;
```

This Ada code is translated into the equivalent Ada code:

```
declare
   L_ENT : LISTE_OF_ENTRIES (1..MAX);
   NB : integer := 0;
   ENT_SEL : ENTRY_NAME;
begin
   if S>0 then
      NB := NB + 1;
      L_ENT(NB) := ENTRY_P;
   end if;
   NB := NB -1;
   L_ENT(NB) := ENTRY_P;
   ENT_SEL := SELECTT(L_ENT (1..NB),
                      -1);
   case ENT_SEL is
    when ENTRY_P => ACCEPTT(ENT_SEL);
    S := S - 1;
    when ENTRY_V => ACCEPTT(ENT_SEL);
    S:= S + 1;
   when others => raise
Tasking_error;
   end case;
end;
```

7. Processor assignement for tasks

In this section, we shall expand on the aspects of processor assignement. The STRAda approach makes it easy to define a generic processor assignement procedure. This generic procedure can be instantiated in the following contexts:

1 - Task control station

This station is able to locate tasks interactively. It provides operators with important information that will enable them to determine their choices (processor loads, assignement of the same processor for tasks likely to communicate with the task to be created, etc.) through an ergonomic man-machine interface.

However, this mechanism is only useful for tasks with long lifetimes (e.g. servers).

2 - Configuration program

The configuration program is written in a separate phase of the main program. It is written in Ada and consists of an assignement function called by the task creation function. This method has the advantage of being valid for programs whatever their lifetime.

3 - Automatic processor assignement manager

The manager frees the programmer from having to worry about processor assignement if he/she so desires. Several managers can be used, each designed for a specific situation: balancing of processor loads, grouping of tasks communicating frequently on the same site, etc..

8. Discussion

8.1. Assessment

In this part, we shall discuss the approach adopted for the STRAda project. As we have already seen, the STRAda approach is above all transformational: we have not attempted to define a new language or system, but simply studied how to transform certain constructs of an existing language into constructs of an existing system. We feel this approach is a good one for several reasons:

- familiar, of the shelf and widely used compilers and systems can be used, thus ensuring good portability;

- it is easily adaptable to other systems: we could also consider transforming constructs for Chorus [28] or for other real-time kernels.

As for installation, we have left aside the problem of shared variables, task termination and time management (*delay* instruction) for the time being. We would stress that with regard to shared variables:

- firstly, certain off-the-shelf systems now provide global memory abstraction over a network;

- secondly, thanks to the minimum STRAda kernel, we can consider a schema in which shared variables are implemented as distributed variables encapsulated in remotely accessible tasks and transformations making it possible to substitute access to these shared variables by calls to entry points defined by the tasks.

The problem of task termination, however, is trickier, as an algorithm adapted to the Ada model is required. This point is being studied.

We also made some implementation choices that sometimes impose limitations on the system. First of all, we chose a UNIX process to implement tasks, even though a thread more closely reflects the nature of a task. The reason for this choice is one of history: threads were not yet supported by UNIX when the Ada project was started. They are still not supported by a lot of systems, and for this reason they would have reduced STRAda's portability.

We also chose sockets instead of RPCs (Remote Procedure Calls). This is related to the choice of Ada rendez-vous as the mechanism for communication between remote tasks: the socket model more closely reflects rendez-vous than the RPC primitive.

Other limitations were imposed on the use of the task model. The COUNT attribute associated with the queue is not implemented as it is impossible to tell the size of the queue on the socket. Flexible arrays are not accepted as a rendez-vous parameter. The same applies to access types, with the exception of task access types, since the task names are replaced during transformation by a single name over the entire distributed environment. Finally, exceptions that are not pre-defined are not fully propagated. When a non pre-defined exception has to be propagated from one site to another, the exception USER_ERROR is raised at the remote site.

8.2. Ada9X and STRAda

Like all other projects studying distribution with Ada, STRAda attempts to provide distributed applications with what was missing from Ada83. The definition of a new Ada standard is near completion. We already know that most of the shortcomings of the current version, with respect to distributed systems, will be fixed by Ada9X. Here, we shall attempt to assess how Ada9X will respond to the needs of distributed applications and to what extent STRAda will continue to be of use once Ada9X has been released.

Unlike Ada83, Ada9X defines a distribution model. The language has a distribution unit, called a partition, and a model for communicating between partitions. On compilation, packages are put into the following classes:

- pure packages;
- remote call interface packages;
- shared passive packages;
- *normal* packages.

It is only at the link editing stage that the partitions are made up by groups of library units. Partitions may be:

- active partitions, which are pseudo-independent programs able to make or receive remote calls;

- shared passive partitions, used to manage data in shared memory areas.

The application programmer has to provide the partition communication sub-system interface for transferring messages between partitions. The compilation system uses this interface to create

parameter passing stubs automatically. This means that a standard distribution model can be used without being dependent on a special type of distribution environment. This model solves the problems encountered when attempting to design distributed applications with Ada83, for two reasons:

- portability is improved by the use of a standard distribution model; only the body of the standard communication sub-system interface RPC_SUPPORT needs to be supplied when changing to another distributed environment;

- static and dynamic reconfiguration is simpler in Ada9X than in Ada83, as in Ada9X the software is not assigned to specific hardware until after code has been compiled.

Static reconfiguration with Ada9X is achieved simply by changing the partitioning specifications or by allocating processors so that the distributed program can be run on a different configuration. For dynamic reconfiguration, we need to be able to create dynamic links between partitions. This is done either by using subroutine access types or tagged class access types.

Exceptions are used in inter-partition communication. An exception raised on calling a remote subroutine is propagated to the caller program. Similarly, a COMMUNICATION_FRROR exception is raised in any exceptional circumstances caused by a network failure or a remote hardware failure.

The Ada9X distribution model is completely independent of the parallelism model. This makes implementation easier, as the model chosen is well suited to distributed systems. On the other hand, in applications where distribution is required only to improve response times or for more efficient use of hardware, it would be useful to be able to free the programmer from distribution issues. This is where STRAda may continue to be helpful once Ada9X has arrived. STRAda makes it possible to distribute any parallel unit, meaning that a distribution system can remain totally transparent.

9. Conclusion

A prototype of the STRAda project is currently operational. We have tested several standard parallel programs, like an adaptation of the dining philosophers' problem in a physically distributed environment.

A practical aspect that we feel is of interest concerns transformations for existing real-time kernels or operating systems; this would make it possible to write or reuse applications written in a high-level language, and to reuse existing and dedicated real-time kernels for certain types of architecture.

We think that STRAda provides some solutions to the problems of distribution with Ada; this will still be the case when Ada9X has been released, since the approach used for STRAda is a complement to that used for Ada9X.

Finally, from a theoretical point of view, it would be interesting to validate the corresponding transformations.

In conclusion, the STRAda project has allowed us to research and establish links between different areas of information technology, by working on both system and language aspects. This experience proved to be very rewarding.

Bibliography

[1] C. Atkinson and A. Di Maio
From DIADEM to DRAGOON
Distributed Ada: developments and experiences, pages 105-136,
Cambridge University Press 1990

[2] A. Ardö and L. Lundberg
The MUMS Multiprocessor Ada Project
Distributed Ada: developments and experiences, pages 235-258,
Cambridge University Press 1990

[3] C. Atkinson, A. Di Maio and A. Natali
Ada for distributed systems
Cambridge University Press 1988

[4] M.J. Bach
The Design of the UNIX Operating System
Prentice-Hall International Editions 1986

[5] G. Bazalgette, D. Bekele, M. Filali, C. Bernon, J.M. Rigaud & A. Sayah
STRAda: Un système de transformations et de répartition pour le langage Ada.
Fourth International Conference, Software Engineering & its applications, 1991, Toulouse, France, proceedings p. 309

[6] G. Bazalgette, D. Bekele, C. [illegible]non, M. Filali, J.M. Rigaud[illegible] A. Sayah
STRAda: An Transformation and Distribution Sy:
Ada: Moving To[illegible]ds 2000, 11th Ada-Europe International Conference
Zandvoort, The Netherlands, June 1992, Proceedings
Springer-Verlag Heidelberg 1992

[7] M. Boari, S. Crespi-Reghizzi, A. Dapra F. Maderna A. Natali
Multi-microprocessors Programming Techniques: MML, a new set of tools
IEEE computer, Jan 1984

[8] J.M. Bishop and M.J. Hasling
Distibuted Ada - the Issues
Distributed Ada: developments and experiences, pages 1-14,
Cambridge University Press 1990

[9] G. Booch
Software Engineering in Ada
Benjamin/Cummings Publishing Company, Inc., Menlo Park, Californie

[10] R. Dewar, S. Flynn, E. Schonberg and N. Shulman
Distributed Ada on Shared Memory Multiprocessors
Distributed Ada: developments and experiences, pages 222-234,
Cambridge University Press 1990

[11] P. Deransart, M. Jourdan, B. Lorho
Attribute Grammars: Definitions and Bibliography.
Springer Verlag, aug 1968

[12] A. Evans, K.J. Butler, G. Goos, and Wulf W.A.
DIANA reference manual. Technical report TL 83-4.
Tartan laboratories incorporated, feb 1983

[13] G. Eisenhauer and R. Jha
Honeywell Distributed Ada - implementation
Distributed Ada: developments and experiences, pages 153-176,
Cambridge University Press 1990

[14] A.D. Hutcheon and A.J. Wellings
The York distributed Ada Project
Distributed Ada: developments and experiences, pages 67-104,
Cambridge University Press 1990

[15] R. Jha and G. Eisenhauer
Honeywell Distributed Ada - Approach
Distributed Ada: developments and experiences, pages 137-157,
Cambridge University Press 1990

[16] R. Jha, M. Kamrad II, and D.T. Cornhill
Ada program partitioning language: a notation for distributed Ada programs.
IEEE Transitions on software engineering, 15(3):271-280, mar 1989

[17] M.B. Jones and R.F. Rashid
Mach and machmaker: kernel and language support for object oriented distributed systems.
In N. Meyriwitz, editor, OOPSLA Proc. of object-oriented programming systems, languages and applications, pages 67-77. ACM, sep 1986

[18] B. Krieg-Brückner and D.C. Luckham.
Anna: towards a language for annotating Ada programs.
SIGPLAN, 30(3):228-238, nov 1980

[19] F. Kordon, P. Sens
Répartir des programmes Ada sur un ensemble homogène de machines UNIX, une expérience de réalisation
Ada, premier bilan d'utilisation, pages 123-135,
Congrès Ada France nov 1991.

[20] J.C. Knight, J.I.A. Urquhart
On the implementation and use of Ada on Fault tolerant Distributed Systems
Ada letters 1984, 4(3), pp. 53-64

[21] B. Liskov
On linguistic support for distributed programs
IEEE Transactions, 8(3)

[22] Bertrand Meyer
Conception et programmation par objets
InterEditions, Paris, 1990

[23] Manuel de Référence du langage de programmation Ada
Alsys, fev 1987

[24] D.T. Ross, J. B. Goodenough et C.A. Irvine
Software Engineering: Process, Principles and Goals

Computer, Mai 1975: 65

[25] T.W. Reps and T.Teitelbaum
The Synthesizer Generator, A System for Constructing Language-based Editors.
Springer Verlag, 1989

[26] T.W. Reps and T.Teitelbaum
The Synthesizer Generator Reference Manual.
Springer Verlag, third edition,1989

[27] A. Volz, P. Krishnan and R. Theriault
Distributed Ada: a Case Study
Distributed Ada: developments and experiences, pages 15-57,
Cambridge University Press 1990

[28] H. Zimmermann, J. S. Banino, A. Caristan, M. Guillemont and G. Morisset
Basic concepts for the support of distributed systems: The Chorus Approach
IEEE Catalog NO. 80-83218, pages 60-67, Apr 81
Computer Society Press

[29] B. Dobbing, I. Caldwell
A Pragmatic Approach to Distributed Ada for Transputers
Distributed Ada: developments and experiences, pages 200-221,
Cambridge University Press 1990

Authors:

C. Bernon and D.Bekele are preparing their PHD at the Paul Sabatier University in Toulouse, France. M. Filali is a research worker at the CNRS (Centre National de Recherche Scientifique). J.M. Rigaud and A. Sayah are assistant professors at the Paul Sabatier University. The domain of interest of the authors are distributed systems, Ada and object oriented languages.

Software Re-Engineering Panel

Moderator: Jim Hohan, Naval Air Warfare Center

Panellists: Jack Cooper, Anchor Software Management
Jay Crawford, Naval Air Warfare Center
John LeBaron, US Army CECOM
Richard Vener, Comptek Federal Systems
Doug Waugh, SEI-SWAP

Program Reuse Experience

Moderator: Dr. Dennis Ahern, Westinghouse Electronics Systems Group

Panelists Session 1: Larry Lang, F-22 SPO
Marcus Musgrave, Lockheed
James Williamson, Wright Laboratory
Ed Beaver & Phil Johnson, Westinghouse

Panelists Session 2: Robert Giordano, PEO, Army Command & Control
Randy Korich, ATTCS CASS
Sandra Zucker, GE Aerospace
Harry Joiner, Telos Systems Group

SEI Measurement Program

Moderator: Dr. Robert Park, Software Engineering Institute

Panelists: Judy K. Fleming, IBM Federal Systems Company
Marshall Potter, Naval Information Systems Mgmt Center

Existing Re-engineering Tools & Capabilities

Moderator: Hans Mumm, NRaD

Panelists: Romel Rivera, Xinotech Research, Inc.
Evan Lock, Computer Command and Control Company
Alex Blakemore, Genoa Software Systems
Wid Spalding, Dynamics Research Corporation
Buck Rizul, Mark V Systems

Futures Panel

Moderator: Miguel Carrio, MTM Engineering, Inc.

Panelists: Dr. Samuel Metters, CEO, Metters Industries
James Smith, President & CEO, SEMA Corporation
Heinz Kagerer, Vice President, SIEMENS-NIXDORF

MATHEMATICS, ENGINEERING, AND SOFTWARE DEVELOPMENT

Michael J. Lutz

Rochester Institute of Technology
Rochester, New York

Abstract: While modern engineering practice owes much to both science and mathematics, it is not identical with either. It is the thesis of this paper that failure to recognize the differences between mathematics on one hand, and engineering on the other, is the root cause of the failure to transfer computing theory into industrial practice. Only by understanding the nature of the engineering method and its relationship to mathematics can we hope to infuse rigor into software development, and in turn transform a craft into an engineering profession.

Introduction

A recurring theme in discussions of software practice is the need to provide a more "scientific" foundation for the field. In part this viewpoint is based on the patent success of science in advancing modern engineering. One obvious conclusion is that software development requires a similar rigorous, mathematical foundation before it can truly be called software engineering.[1, 2] Supporters of this position are disconcerted by the fact that theoretical results have little effect on the state of the practice.

It is my thesis that at the failure of theory to influence practice is rooted in a lack of appreciation of the fundamental differences between science (specifically, mathematics) and engineering. Indeed, I believe that at least some of the resistance to formalism is based on sound engineering principles, however poorly articulated.

Many of those advocating increased rigor, myself included, were trained as mathematicians or scientists, not as engineers. Given this, it is important to investigate the methods of engineering and their relationship to science and mathematics. Without such an understanding, attempts to apply theory to the engineering of software are naive at best, and counterproductive at worst. However, armed with such knowledge, it will be easier to spot areas where theory can have the greatest positive effect.

I have chosen the field of formal methods to exemplify both the problems and promises of theory as applied to practice. While this is the area with which I am most familiar, I firmly believe that similar comments apply to other applications of mathematics and science. For instance, a recent book on software reliability directly addresses the needs of practitioners.[3]

The next section provides a brief introduction to the world of formal methods, providing a broad sketch of the method types, as well as the role each type may play in software development. A discussion of the engineering method follows, with particular attention paid to the role of science. This leads naturally to a discussion of the use of formal methods in the context of the engineering method. In particular, a strategy is proposed for the effective transfer of formal methods to standard practice.

An Overview of Formal Methods

Any discussion of formal methods requires a definition of this term:

> A *formal method* is a mathematical system and associated rules of inference used to *model* and *reason about* some characteristic of a software system.

While many such systems are applicable to software development, the presentation will concentrate on those describing functional behavior. This is where the majority of work to date has taken place, and where, as a consequence, the mathematical models are most mature.

Even with this restricted focus, however, further categorization is desirable. Perhaps the clearest distinc-

tion is between the *model-based* approaches such as Z[4] and VDM,[5] and *algebraic* (equational systems) such as OBJ.[6] Model based systems use a collection of mathematical entities (sets, relations, sequences, etc.) as building blocks to construct problem-specific models. Algebraic systems are based on equations relating values drawn from different *sorts*. As a rule, algebraic systems are used primarily to define the semantics of abstract data types (ADTs), whereas model based theories are employed for system level descriptions. Of course, there is large overlap in the problems to which these two approaches apply.

Competing methods can also be classified by the software development phases where they are most useful. The Z system, for example, is most frequently used for system specification, though approaches exist to carry it through design and implementation. Similarly, VDM also spans the specification and design phases, though its emphasis is clearly on the latter. Finally, Dijkstra's guarded command language[7] is primarily for proofs of (implementation) correctness, using a formal design specification already at hand.

The relative importance of modeling vs. reasoning provides a third perspective. All formal methods combine modeling and reasoning, yet there is a discernible shift in emphasis from the former to the latter as activities proceed from analysis to implementation. Z and VDM are often presented as tools for precise and concise modeling of a system or its components. In addition, they are also used to verify design steps, by showing the formal design model mirrors the formal specification in all essential properties. At the implementation stage, where a system like Dijkstra's is applicable, the system has been partitioned into manageable sized components, each of with a precise specification. Here the modeling activity is implicit, while the inference rules highlight the role of formal reasoning in simultaneously creating algorithms and proving their correctness.

A final role that formal methods can play is in cataloging systems of reusable components. The analogy here is to a standard IC component catalog; an example would be using the two-tiered Larch system[8] to describe interfaces of C modules. By providing complete, accurate, and concise descriptions of library components, formal methods help developers evaluate the fitness of components for their application. Note that this use of formal methods is primarily one of modeling.

There is a common thread through all of the viewpoints above. In all cases, mathematics (specifically discrete mathematics) is used to characterize the behavior of a software product. The question remains whether or not these systems are useful as engineering tools. To answer this question, we must first determine what constitutes engineering, and how mathematics supports engineering endeavors.

The Engineering Method

If engineering disciplines are to serve as guides for software development, then an investigation of the *engineering method* is in order. That is, the focus must be on the sorts of problems that require engineers, and how engineers approach such problems. We are less concerned with the goals of engineering than with the processes by which engineers achieve these goals. With respect to formal methods, the question is even more specific: what is the role of mathematics and science in the everyday activities of practicing engineers? The answer should help determine if and when formal methods can be useful in software development.

A problem with this investigation is that most engineers are not introspective by nature. In addition while the philosophy of science is a respectable academic discipline, similar work on the philosophical foundations of engineering is rare indeed. Yet without such a perspective, it is presumptuous for non-engineers to speak of *any* practice as promoting software engineering. Fortunately, there are professional engineers who have written about their profession and its workings.[9, 10, 11]

A primary source for my investigations is a short work by Billy Vaughn Koen.[11] In this 75 page monograph, Koen tries to define the engineering method in a manner that is acceptable to professional engineers and accessible to the public at large. In Koen's view, the engineering method is:

> The strategy for causing the best change in a poorly understood or uncertain situation within the available resources.

To an engineer, *best* is not absolute, but relative to the engineer's ability to assess and balance both technical and societal issues. Thus professional engineers frequently disagree over proposed designs, but primarily on the basis of which aspects are significant and which can be dismissed as inconsequential.

Koen then asserts that the engineering method can only be understood as the application of appropriate

engineering heuristics. He defines the "state-of-the-art" (or *sota*) as the heuristics available to the engineering profession at a specific point in time. Each engineer has his or her own sota, which reflects the heuristics he or she can bring to bear on a problem. An engineer engages in *best practice* to the extent that his or her individual sota intersects with that of the profession as a whole.

Koen draws a strong contrast between this pragmatic approach and the processes of science. Two conflicting scientific theories cannot both be correct: at least one is wrong. The victorious theory in any scientific debate is the one which explains a broader spectrum of phenomena, or which provides a simpler model of the same phenomena.

Engineering heuristics, however, are never right or wrong. Instead, they are more or less applicable in a specific context. Indeed, many engineering heuristics are discarded scientific theories that prove sufficient for the task at hand (e.g., Newtonian physics). What is more, the engineering sota can accommodate conflicting heuristics, with higher level heuristics to determine when each is used.

There are two key consequences of Koen's perspective. First, engineering practice has no absolute measure for evaluation, if for no other reason than the sota is undergoing constant change. Second, the heuristics used in a particular case depend on more than purely technical factors: societal norms, resource constraints, and evaluation of risks. Koen mentions the Golden Gate Bridge, which is not really made of gold despite that metal's advantageous properties.

Thus, in Koen's view, a concept, technique, or approach is judged by its relevance to the task at hand. And, of course, as judgement is involved there is always a chance things may go sour. What distinguishes engineering is the evolution of heuristics that succeed much more often than they fail.

Koen discusses a variety of heuristics are various levels of detail. Some are quite specific to a branch of engineering; those for software, for instance, include avoiding gotos and writing procedures that will fit on one screen. However, there are also higher-level heuristics shared by most engineering disciplines:

- Work at the margin of solvable problems (don't stretch too far beyond what is known to work).
- Make small changes to the state-of-the-art (don't make radical changes to the process or the product).
- Allocate resources as long as the cost of not knowing exceeds the cost of finding out (expend resources to evaluate and reduce risk).

Many of these are directly applicable to software development. Firms that make radical, uncontrolled process changes often regret their impetuousness. Similarly, prototypes are promoted as a way of reducing the risk that delivered products will not satisfy the customer.

One of Koen's heuristics discusses the use of science (and by extension mathematics). Koen draws a sharp distinction between applied science on one hand and engineering on the other:

> The thesis that engineering is applied science fails because scientific knowledge has not always been available, is not always available now, and because, even if available, it is not always appropriate for use.
>
> ...
>
> The engineer recognizes both science and its use as heuristics, although very important ones, to be applied only when appropriate.

Examples abound of scientific and mathematical theories that took decades, even centuries, to be incorporated into engineering practice. Many applications of Fourier analysis, for example, had to await the development of digital computers and the dissemination of the Cooley-Tukey FFT algorithm. Similar observations may be made about the use of finite element analysis in mechanical design.

In summary: engineers do not ignore problems simply because scientific theory is lacking, nor do they blindly apply all scientific results as they appear. Science is a powerful tool for engineering, but the tool's use is guided by heuristics that also consider time, cost, and other resource constraints.

While this discussion has been based on Koen's work, his opinions are supported by other commentators on the engineering scene.[9, 10, 12, 13] In all cases, the critical issue in engineering, and the focus of engineering creativity is on *design*, which can only be understood in the context of *engineering judgement*. This judgement is co-terminus with the discriminating use of heuristics, which in turn provides a useful reference point when considering the application of computing theory in software engineering. The particular question we must address is how to increase the value of the heuristic "use formal methods."

Formal Methods and Software Engineering

It is safe to say that use of formal methods is not common in industry. The previous section gives us clues as to why this is so. First, formal methods are a dramatic departure from current practice, and this is counter to the "small change" heuristic. Second, while formal methods are elegant science, most practitioners do not believe they are cost-effective. A recent book contains a scathing attack on formalism on just such grounds.[14]

There are many reasons for such skepticism, not all of which are easily dismissed. Part of the problem is the emphasis (at least in the U.S.) on formal program proofs. Not only do such proofs appear tedious and time-consuming, they presuppose a formal specification without describing how such specifications are themselves developed. What is more, such proofs occur at the implementation stage of the development life-cycle, which consumes a relatively small proportion of a project's resources. Finally, proofs do little to increase confidence that the specifications are correct, or that the design reflects all requirements. Yet is is increasingly clear that specification and design errors are both more expensive to repair and more pernicious in their effects than those introduced during implementation.

With these criticisms in mind, and remembering the key tenets of the engineering method, it is feasible to consider how formal methods might be most effectively used in software development. The following sections sketch a three-phase strategy to increase the acceptance and application of formal methods.

Emphasis on Modeling

As mentioned above, it is skepticism about proofs (i.e., formal reasoning) that dominates discussions of formal methods. One way to address this issue head-on is to concentrate on the modeling aspect. First, it is easier to develop and read formal specifications than it is to reason about their properties. Indeed, it has been argued that using mathematics as a descriptive tool is valuable in and of itself, as the result is increased clarity and reduced ambiguity.[15, 16] Second, it is easier to gain acceptance for a new technology when it addresses front-end problems.

The prerequisite mathematics are the same for both formal specifications and proofs of correctness. However, the level of description is higher and the potential for improved quality more obvious in the case of specifications. Once the use of formalism as a descriptive tool is accepted, it becomes much easier to incrementally address issues of formal consistency analysis and verified design. This may, in time, lead all the way to formal proofs of some components. Note, however, that in all cases the decision of whether or not to formally verify a particular component remains heuristic.

The most frequently cited case of industrial formal methods is the CICS reengineering work at IBM's Hursley Labs in the United Kingdom, performed in conjunction with Oxford University.[17] In this case, Z is used to model the CICS components being reengineered. While refinements to Dijkstra's language are part of the process, it appears that the modeling component is most important. In particular, Z is now being used to provide precise interface specifications for CICS application developers.

Consumable Mathematics

Another impediment to increased use of formal methods is the level of mathematical sophistication required. In most engineering disciplines, practitioners become master users of high level mathematical and scientific results. Engineers use handbooks and other reference material to access the essence of a theory; deriving necessary information from scratch is unusual. Even in cases where engineers must apply fundamental principles, they do so at an abstract level (e.g., using general differentiation and integration rules).

Most formal methods, however, require intimate and detailed knowledge of fundamental mathematical theories. This may be a natural (if regrettable) consequence of the immaturity of these methods. Still, the fact remains that engineers rarely resort to the limit definition of differentiation. Similarly, we should not expect software engineers to routinely employ the low-level inference rules of natural deduction.

An example of providing such higher level abstractions is the work on instrumentation models at Tektronix.[18] The creation of such models requires a deep knowledge of Z semantics. Development engineers, however, can simply employ these models, having been assured that they rest on firm foundations.

Engineers are not and should not be mathematicians. Instead, the formal methods community must develop higher-level "consumable mathematics" corresponding to continuous mathematics for traditional engineering. It is heartening to see that work is progressing on this front.[19, 20]

Education of Future Professionals

The role of education in successful technology transfer is vital. There are many instances where technology introduced in the classroom supported significant change in industry within a few years. The success of the Unix system, at least in the engineering workstation marker, is in large measure due to the wide-scale use of the system by academia in the 1970's. It does not matter whether one thinks this was good or bad; what is significant is that academic experiences led to expectations that were eventually met by industry.

Given this, it is crucial that students be exposed to formality in such a way that they consider it an important component of their intellectual toolkits. If this is not so, there is a risk that the next generation of developers will be as skeptical as their predecessors about formal methods. Thus, the prescription above is as valid for software engineering curricula as it is for industrial acceptance. That is, instructors should stress modeling early, postponing formal proofs until such time as their significance can be demonstrated. The result: more graduates who appreciate what formal methods have to offer and who can articulate these advantages during their professional careers.

I have adopted such a descriptive approach in two courses I teach at RIT. The first course, an introduction to design and implementation for sophomores, uses simple pre and post conditions to define the contract between clients and implementors of modules. In addition, implementors are required to develop invariants that characterize the legal internal states of their modules. While our teaching language is Modula-2, the spirit is that of CLU-based work at MIT.[21] I consider this technique successful, as many students have carried these practices over to later courses taught by others.

The second course is on specification and design, taken by juniors and seniors in the software engineering concentration. In this course, I introduce modeling in Z, using the text by Potter, et. al.[22]. Later, when object-oriented technology is presented, I incorporate Z as a functional specification tool, using the work of Hayes and Coleman on coherent object-oriented analysis.[23] The notion of design refinement and functional verification is touched upon, but always in the context of modeling a problem solution. The results are gratifying: surveys at the beginning and end of the course show a definite shift towards the acceptance of formality as useful in practice. Though the students are skeptical about their ability to transfer this technology to industry, the seeds of this transfer have been planted.

Conclusion

A formal, scientific foundation is required if software development is ever to be classified as engineering. However, developers of this foundation must be aware of the processes underlying engineering endeavors if the resulting theory is to become part of standard practice. Based on an investigation of the engineering method, this paper has proposed one strategy for achieving such integration with respect to formal methods. Similar strategies must be defined for other areas of computer science so that a modern software engineering discipline can emerge.

Acknowledgements

My views on engineering and its relationship to software development have been honed and sharpened by my work with dedicated engineers in industry, and by thoughtful and provocative discussions with associates in RIT's College of Engineering. My friend and colleague Henry Etlinger has provided the positive criticism that helped me clarify my positions. Finally, Suzanne Bell of RIT's Wallace Memorial Library has been of invaluable assistance in helping me track down relevant materials in the areas of science, engineering, and technology transfer.

References

1. Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, 7(6):15–24, November 1990.

2. Anthony Hall. Is software engineering? In C. Sledge, editor, *SEI Conference on Software Engineering Education*, number 640 in LNCS. Springer-Verlag, October 1992.

3. John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software Reliability : Measurement, Prediction, Application (Professional Edition)*. McGraw-Hill, New York, 1990.

4. J. M. Spivey. *The Z Notation: A Reference Manual (2nd Edition)*. Prentice-Hall, Englewood Cliffs, N.J., 1992.

5. Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, Englewood Cliffs, N.J., 1990.

6. Joseph Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, 10(5):528–543, September 1984.

7. Edsger Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

8. John Guttag and James Horning. An introduction to LCL, a Larch/C interface language. Technical Report 74, Digital Systems Research Center, July 1991.

9. Samuel C. Florman. *The Existential Pleasures of Engineering.* St. Martin's Press, New York, 1976.

10. Henry Petroski. *To Engineer Is Human: The Role of Failure in Successful Design.* St. Martin's Press, New York, 1985.

11. Billy Vaughn Koen. *Definition of the Engineering Method.* American Society for Engineering Education, Washington, D.C., 1985.

12. Mary-Frances Blade. Creativity in engineering. In Myron A. Coler, editor, *Essays on Creativity in the Sciences.* New York University Press, New York, 1963.

13. Thomas Allen. Distinguishing engineers from scientists. In Ralph Katz, editor, *Managing Professionals in Innovative Organizations.* Ballinger Publishing, Cambridge, MA, 1988.

14. Nathaniel S. Borenstein. *Programming as if People Mattered: Friendly Programs, Software Engineering and Other Noble Delusions.* Princeton University Press. Princeton, New Jersey, 1991.

15. Darrel C. Ince. *An Introduction to Discrete Mathematics and Formal System Specification.* Oxford University Press, 1988.

16. Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–20, September 1990.

17. Paul Johnson. Experience of formal development in CICS. In John A. McDermid, editor, *The Theory and Practice of Refinement*, pages 59–78. Butterworths, London, 1989.

18. Norman Delisle and David Garlan. A formal specification of an oscilloscope. *IEEE Software*, 7(5):29–36, September 1990.

19. John Guttag, James Horning, and Andres Modet. Report on the Larch shared language. Technical Report 58, Digital Systems Research Center, April 1990.

20. Darrell Ince and Derek Andrews. *Practical Formal Methods with VDM.* McGraw-Hill, 1991.

21. Barbara Liskov and John Guttag. *Abstraction and Specification in Program Design.* MIT Press, Cambridge, 1986.

22. Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

23. Fiona Hayes and Derek Coleman. Coherent models for object-oriented analysis. In Andreas Paepcke, editor, *OOPSLA '91.* ACM Press, October 1991.

Biography

Michael J. Lutz is a Professor of Computer Science at the Rochester Institute of Technology in Rochester, NY, where he has been a faculty member since 1976. In addition, he has industrial experience as a software designer, project leader, group manager, and consultant in areas as diverse as portfolio asset management, real-time operating systems, optical metrology, data communications, microfilm imaging, and environmental auditing. His research interests include object-oriented technology, formal methods, and computer science education. Professor Lutz received a B.S. in Mathematics from St. John Fisher College and an M.S. in Computer Science from SUNY Buffalo. He is a member of the ACM and the IEEE Computer Society. He can be contacted via electronic mail at **mjl@cs.rit.edu**.

MODELING THE COSTS OF MILITARY SOFTWARE

Capers Jones, Chairman
Software Productivity Research, Inc.
1 New England Executive Park
Burlington, MA 01803
617 273-0140

Summary

Software produced for the U.S. military services is one of the key components of national defense, and will play an increasing role in all future military operations. It is therefore of critical importance to understand the key factors which influence the costs, schedules, and quality levels of software production.

Unfortunately, historical measures based on "Lines of Source Code" have tended to conceal vital information, and have slowed software research efforts. Improved metrics based on the functional content of software are now available. These new metrics reveal that coding itself is not the major cost driver of large-scale software production. Both paperwork and defect removal costs outweigh pure coding by substantial margins for military software.

Introduction

Measurement, metrics, and statistical analysis of data are the basic tools of science and engineering. Unfortunately, the software industry has existed for almost 50 years with a dismaying paucity of measured data, with metrics that have never been formally validated, and with statistical techniques that are at best questionable.

As the 20th century draws to a close, it is desirable for the phrase "software engineering" to cease being an oxymoron, and become a valid description of a true engineering discipline. An important step in that direction will be to evaluate and validate all of the common metrics and measurement techniques for software under controlled conditions, in order to select a standard set that can be used with little or no ambiguity and with minimal subjectivity.

Consider some of the basic metrics that confront us in a normal day. Upon arising, we may check the outside *temperature* and perhaps note the *barometric pressure* to judge if it will rain. At breakfast, we may think about the *cholesterol levels* and the *calories* in the food we eat. If we purchase gasoline on the way to work, we perhaps consider the *octane rating* of the fuel we are purchasing. We might also reflect on the *horsepower* of the automobile engine.

All of the above metrics are interesting in several respects: 1) They are synthetic metrics which deal with invisible phenomena

that cannot be counted directly; 2) Most adult humans understand the significance and basic meaning of the metric; 3) Very few individuals (less than 1%) humans know how to calculate these metrics or understand their mathematical derivations.

Now consider the metrics history of software engineering. Prior to the 1970's, the phrase "software engineering" did not exist. Those of us in the field were simply called "programmers."

The only metrics that we used were simple natural metrics such as integers. We tended to use *lines of code* because we wrote our programs on special programming tablets where the lines were clearly visible and often numbered. Some programming languages such as Basic could not even be used without line numbers.

In the 1960's and 1970's, fairly low-level languages such as Assembly, JOVIAL, and FORTRAN were being used for the bulk of military programming. The effort devoted to programming and code-related work was the dominant activity of software. Other activities such as requirements, design, and user documentation were seldom measured at all.

When non-coding activities were measured, we used simple natural metrics such as integer counts of the pages or words. Graphics seldom occurred in commercial software, and topics such as reusable code, object-oriented languages, pull-down menus, graphical interfaces, mice, etc. were still in the future.

In the 1980's and early 1990's, an explosion of new languages, new methods, and new approaches changed the work of software development in profound ways.

By the start of the 1990 decade more powerful languages such as Ada, Objective C, C++, and Modula 2 were being used. New CASE tools and I-CASE tools were available, which offered significant new capabilities to software engineers. New standards such as DoD 2167A were in effect for military software, and the ISO 9000 standard series was starting to be deployed for civilian software.

These new approaches made profound changes in the cost and effort structure required for software production and maintenance. The tasks associated with pure coding were being reduced almost daily. However, the tasks associated with planning and specification preparation were increasing daily as well.

Unfortunately, attempts to model these profound changes in the software paradigm using traditional "Lines of Source Code" metrics discovered deep mathematical problems with the metric itself. LOC metrics were proved to have a built-in bias which penalized more powerful languages such as Ada. LOC metrics also failed to deal with the enormous costs and resources devoted to plans, specifications, and other forms of software paperwork.

The first powerful synthetic metric developed for software was the Function Point, which was created in the middle 1970's by Allan Albrecht and his colleagues at IBM. This metric was placed in the public domain in October of 1979 at a joint SHARE/GUIDE/IBM conference in Monterey, California (1).

If future historians want to explore the evolution of software engineering as a true engineering discipline, October 14th 1979 is a strong contender to be considered the exact starting point. Allan Albrecht's presentation in Monterey marks the first day in software history than an effective synthetic metric for software was publicly stated.

Problems with "Lines of Source Code" Metrics

The subjectivity of "Lines of Source Code" can be illustrated by the following analogy: Ask a software engineer or software manager a basic question: "Is the speed of light the same in the United States, and Germany?" Obviously the speed of light is the same in every country.

Then ask the following question: "Is a Line of Source code the same in the United States and Germany?" The answer to this question is, "No, it is not"

Software articles and research in Germany has tended to use physical lines more often than logical statements, while the reverse is true for the U.S. and Japan.

There have been other metrics that differed from country to country, such as U.S. gallons and Imperial gallons. Also statute miles and nautical miles differ significantly. These differences are common knowledge, while the differences in "Lines of Source Code" definitions are comparatively obscure, and sometimes not fully stated by software authors.

The most widely used software metric since the industry began has been Lines of Source Code. or LOC. Either this metric or "KLOC" (where K stands for 1000) have been used in print in more than 10,000 articles and books since 1946 Most users of LOC and KLOC regard this metric as being objective, and indeed a number of standard reference books and articles on metrics have cited the objectivity of LOC as a key virtue.

However, from discussions with more than a thousand software managers and professionals, it is unfortunate to report that the LOC metric may be the most subjective metric used in refereed articles in the last 50 years.

When LOC and KLOC originated as software metrics, the only languages in use were machine language and basic assembly language. For basic assembly language, physical lines and logical lines were equal: each source statement occupied one line on a coding sheet or one tab card. From 1946 until about 1960, LOC and KLOC metrics were reasonably well defined and reasonably objective. The explosion of languages from 1960 forward destroyed the objectivity of LOC and KLOC, and their validity for economic studies as well.

From surveys of counting practices carried out by the author and his colleagues at Software Productivity Research, the varieties of subjective methods associated with LOC counting creates a total range of apparent size of more than one order of magnitude for the software industry as a whole. The largest number of major code counting variations observed within a single company was six, and the range for counting the size of a single control project within that company was approximately 5 to 1. This is far too broad a range to be tolerated for an engineering metric.

The standard dictionary definition of subjectivity is "Particular to a given individual; personal." Under that definition, it must be concluded that LOC and KLOC are in fact subjective metrics and not objective ones.

Code counting subjectivity could be eliminated by establishing standard counting conventions for each major language. Indeed, Software Productivity Research (2), the IEEE (3), and the Software Engineering Institute (4) published preliminary draft LOC counting proposals within a year of one another.

Unfortunately, the SPR, IEEE, and SEI draft standards differ, so even in the domain of standardization of LOC counting practices subjectivity is present. Note that for many modern languages such as 4GL's, spreadsheets, query languages, object-oriented languages, and graphics icon-base languages, none of the current draft standards are technically acceptable.

The LOC Paradox

The LOC and KLOC metrics have a much deeper and more serious problem than a simple lack of standardization: LOC metrics are troubled by a deep mathematical paradox. Both productivity and quality appear to move backwards when measured with LOC!

Indeed, the tendency of LOC and KLOC to move backwards as economic productivity improves is a much more serious problem for software economic studies than the subjectivity of LOC and KLOC. This mathematical problems with LOC are severe enough so that they make the phrase "software engineering" seem ridiculous. It is embarrassing for a major industry such as software to continue to use a metric that does not work, and to do so without even realizing what is wrong with it!

Unfortunately, many well-known books on software measurement and economics do not contain even a single statement about this well-known problem. To cite but two examples, both Barry Boehm's Software Engineering Economics (5) and Robert Grady's and Deborah Caswell's Software Metrics: Establishing a Company-Wide Program (6) use LOC and KLOC metrics without any warnings or cautions to the readers of the paradoxical nature of these metrics for high-level languages.

Unfortunately, the software measurement initiatives at SEI (7) also fail to discuss the problems and paradox of LOC metrics, and do not discuss functional metrics at all. These unfortunate omissions place the SEI measurement work some distance behind the state of the art, although other aspects of the SEI measurement studies are fairly advanced.

The paradox with LOC and KLOC is caused by the impact of the fixed and inelastic costs of certain activities that are always part of software projects. The problem of measuring productivity in the presence of fixed costs has long been understood for manufacturing economics.

However for software, it was initially described by the author in 1978 (8), and fully explained in 1986 in the book Programming Productivity (9).

There is a basic law of manufacturing economics that if a manufacturing process includes a high percentage of fixed costs, and

the number of units produced goes down, the cost per unit will go up. This same law also applies to software. When LOC is used as a manufacturing unit, and there is a move from low-level to high-level languages, then obviously the number of "units" to be created will decline in the presence of fixed costs.

Using LOC and KLOC metrics for a single language can produce valid results if standard counting rules are applied. However, for cross-language comparisons, or for projects containing multiple languages (such as Ada and Assembly) the results are always invalid and paradoxical.

The LOC metric, compared to Function Points, also distorts quality measurements. The situation with LOC is so paradoxical and absurd in the presence of high-level languages that it is fair to state that the LOC metric has slowed the advance of software engineering as a true engineering discipline. It is time to step up to this problem, and declare LOC metrics to be an example of professional malpractice.

Malpractice is a serious situation, and implies the usage of an approach known to be harmful under certain conditions, which should have been avoided through normal professional diligence. For example, a medical doctor who prescribed penicillin for a patient known to be allergic to that antibiotic is an illustration of professional malpractice. Using LOC and KLOC metrics to evaluate languages of different levels without cautioning about the paradoxical results that occur is unfortunately also an example of professional malpractice.

The LOC and KLOC metrics grow progressively more ambiguous and counter-intuitive as the level of languages goes up or for multi-language studies. Following are situations where LOC and KLOC are ambiguous enough to be harmful to economic understanding and their usage should constitute malpractice:

A) LOC and KLOC metrics should be avoided for economic studies involving object-oriented languages, 4GL's, generators, spreadsheets, and graphic-icon based languages.

B) LOC and KLOC metrics should never be used to compare unlike languages, such as C++ and Ada.

C) LOC and KLOC metrics should not be used for applications containing multiple languages, such as C and Assembly or Ada and Assembly.

D) LOC and KLOC metrics should not be used to measure software plans, specifications, or other non-code deliverables.

E) LOC and KLOC metrics should not be used for quality normalization (i.e. defects per KLOC) for studies involving multiple languages.

Consider the similar problem of carrying out international cost surveys that involve multiple currencies such as dollars, yen, pounds, lire, deutschmarks, francs, etc.

There are two methods for carrying out accurate international cost surveys: A) One of the currencies such as dollars is selected as the base currency, and all other currencies are converted into equivalent amounts; 2) A synthetic base metric such as European Currency Units (ECU) is selected, and all quantities are expressed in those units.

The acceptable methods for dealing with multiple currencies provide a useful model for software studies dealing with multiple languages:

A) One of the languages such as Assembly Language is selected as the base language, and all other languages are converted into equivalent amounts.

B) A synthetic base metric such as Feature Points is selected, and all quantities are expressed in those units. Of these two methods for dealing with multiple languages, method B is preferred today.

Comparing Ada and Assembler with LOC

Since the Ada language is such a key component of military software strategy, it is important to understand the way LOC metrics interact with Ada.

Following are two abstract examples of the same project, with one version created in Assembly Language and the other in the Ada language, to clarify the paradox of LOC metrics.

Assembly Language Version
(10,000 LOC and 50 Function Points)

Activity	Effort	Costs
Requirements	2.0	$20,000
Design	4.0	$40,000
Coding	6.0	$60,000
Testing	4.0	$40,000
Documents	2.0	$20,000
Management	2.0	$20,000
Totals	20.0	$200,000

Now consider the same project, only let us assume that the programming language used was Ada:

Ada Language Version
(2,000 LOC and 50 Function Points)

Activity	Effort	Costs
Requirements	2.0	$20,000
Design	2.0	$20,000
Coding	1.5	$15,000
Testing	1.5	$15,000
Documents	2.0	$20,000
Management	1.0	$10,000
Totals	10.0	$100,000

Using standard economic definitions, the Ada version is twice as productive as the Assembly language version, since the same goods were delivered with only half the effort and expense.

However, when productivity is measured using manufacturing economics, with LOC defined as the unit of manufacture, the real economic advantages of Ada cannot be seen:

	LOC per Staff Month	Cost per Source Line
Assembly	500	$20.00
Ada	200	$50.00

When the manufacturing unit is switched from LOC to Function Points, the economic advantages of Ada become clear.

Since both versions perform the same functions, assume that the Function Point totals of the Assembly and Ada versions are identical: 50 Function Points each.

	FP Per Staff Month	Cost per FP
Assembly	2.5	$4000
Ada	5.0	$2000

Observe that when Function Points are used as the unit of manufacture, rather than Lines of Code, standard economics and manufacturing economics now agree. Ada is significantly more productive than Assembly language.

Function Points are synthetic metrics, and one of the advantages of synthetic metrics is that they have wide general utility. For example, the synthetic metric *horsepower* can be used on electric, diesel, and gasoline engines with equal precision.

Natural metrics, such as LOC, cause serious trouble when they are used outside their normal domain. In the case of LOC, it the inclusion of non-coding activities which degrade their accuracy.

Data from the MK-160 Gun System

The basic thesis of this paper is that coding is no longer the dominant cost driver for military software projects, and other elements such as paperwork, testing, and non-coding tasks now constitute the bulk of military software costs.

If coding is only a minor portion of total software costs, then it is inappropriate to use LOC metrics for the entire project. Synthetic metrics such as Function Points are much more appropriate for normalization of mixed-activity economic and quality studies.

It is useful to conclude by examining actual data. A report on the MK-160 gun computing system produced by Paul W. Lusher of the Naval Surface Weapons Center at Dahlgren (10) provides confirmation of the hypothesis that coding is no longer the dominant cost driver for military software applications.

The MK-160 is a mixed-language system written primarily in CMS2 and containing about 120,632 LOC and 1240 Function Points.

The sum of the plans, specifications, and user documents for the project totaled to 5,585 pages.

The total effort for the project was 795.6 person months.

Coding itself constituted 221.5 person months, or only 27.8% of the total.

Activities concerned with paperwork (plans, specifications, user documents) amounted to 256.8 person months, or about 32.3% of the total.

Activities concerned with defect removal operations (reviews, inspections, testing) amounted to 200.1 person months of effort, or about 25% of the total.

The effort for the various non-coding activities associated with this project far outweighed the code-related activities:

about 72.1% of the total effort went on non-coding activities.

Following are some of the details of this project, to illustrate the mixture of coding, paperwork, defect removal, and other activities which comprise modern military weapons software:

Software Effort for the MK-160 Gun System

Activity	Person Months of Effort
Development plan	5.7
Test plan	6.9
Personnel management	91.4
Progress reports	25.4
Configuration control	25.7
Requirements	58.4
System architecture	11.3
Initial specification	65.3
Final specification	23.6
Data design spec	23.1
Data structure review	4.5
Coding	221.5
Unit test	14.6
Function test	21.3
Regression test	7.9
Integration	14.7
Integration test	18.5
Stress test	10.4
System test	31.6
Field test	23.1
Independent test	23.7
Operator's guide	12.2
Maintenance manual	21.9
Reference card	0.2
Total	795.6

The overall productivity rate for this project expressed in LOC is about 152 LOC per person month (note that pure coding had a rate of about 545 LOC per person month).

Expressed in Function Points per person month, the overall rate was 1.56, and the coding itself had a rate of about 5.6 Function Points per person month.

For mixed language projects, and for comparison between projects, Function Points are markedly superior to the older LOC metrics for all normalization, economic, and quality research purposes.

Summary and Conclusions

Software development in 1993 is changing dramatically under the combined impact of new languages, new standards, new tools, and new methods.

In order to explore the impact of these new approaches, it is urgent for the software industry -- both military and civilian -- to be able to measure the impact of improved practices.

Lines of Code metrics are no longer viable, and indeed a case can be made for relegating LOC metrics to the category of "professional malpractice."

Modern functional metrics are becoming the dominant tool for exploring software productivity and quality as the industry matures.

Indeed, the non-profit International Function Point Users Group (IFPUG) has been growing at a rate of 46% per year and

is now the largest measurement association in the United States.

It is critical that the software measurement work of the U.S. military services, the DoD, and the Software Engineering Institute (SEI) be at state of the art levels.

This means that the basic concepts of functional metrics should now be included in the training of software engineers and software managers.

References

1) Albrecht, A.J.; "Measuring Application Development Productivity"; Proceedings of the Joint IBM/SHARE/GUIDE Application Development Symposium; October 1979; pp 83-92.

2) Jones, Capers; "Rules for Counting Procedural Source Code;" Applied Software Measurement; McGraw-Hill; New York, NY; 1991; pp 309-316.

3) Draft Standard for Software Productivity Metrics; P1045/D2.1; IEEE Software Productivity Metrics Working Group; December 1990.

4) Software Size Measurement with Application to Source Statement Counting; Software Engineering Institute (SEI), Pittsburgh, PA; August 1991.

5) Boehm, Barry W.; Software Engineering Economics; Prentice Hall, Englewood Cliffs, NJ; 1981; 767 pages.

6) Grady, Robert B. and Caswell, Deborah L; Software Metrics -- Establishing a Company-Wide Program; Prentice Hall, Englewood Cliffs, NJ; 1987; 288 pages.

7) Rozum, James A.; Software Measurement Concepts for Acquisition Program Managers; Technical Report CMU/SEI-92-TR-11; ESD-TR-92-11; Software Engineering Institute; Carnegie Mellon University; Pittsburgh, PA; June 1992; 68 pages.

8) Jones, Capers; "Measuring Programming Productivity and Quality;" IBM Systems Journal; Vol. 17, No. 1; 1978; pp. 36-63; IBM Corporation, Armonk, NY.

9) Jones, Capers; Programming Productivity; McGraw-Hill, New York, NY; 1986; 282 pages.

10) Lusher, Paul W.; Productivity Baseline Report for MK-160 Gun Computing System Software Development; Gun Fire Control Systems Branch (G72); NAVSWC, Dahlgren, VA; October 1991; 51 pages.

Futures Panel

Moderator: Miguel Carrio, MTM Engineering, Inc.

Panelists: Dr. Samuel Metters, CEO, Metters Industries
James Smith, President & CEO, SEMA Corporation
Heinz Kagerer, Vice President, SIEMENS-NIXDORF

Author's Index